MASTER EN SCIENCES INFORMATIQUES

Étude du langage GUARDED HORN CLAUSES dans le cadre d'applications robotiques

Dardenne, Anne

Award date:
1986

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Année académique 1985-86

Etude du langage GUARDED HORN CLAUSES dans le cadre d'applications robotiques

Anne DARDENNE

Mémoire présenté
en vue de l'obtention du grade de
Licenciée et Maître en Informatique

Avant-propos

Je tiens avant tout à remercier le Professeur Axel van Lamsweerde , promoteur de ce mémoire, pour m'avoir proposé un sujet et un stage qui m'ont donné accès au domaine de l'Intelligence Artificielle. Ses conseils ont permis à ce mémoire d'acquérir la base théorique qui lui était nécessaire.

Je tiens également à remercier Luc Dekeyser, pour m'avoir accueillie dans le service d'Informatique Médicale de l'hôpital Gasthuisberg à Leuven, et avoir ainsi rendu mon stage possible. Son enthousiasme pour les recherches effectuées fut à l'origine de bien des idées nouvelles indispensables à l'avancement du travail.

Les membres du service d'Informatique Médicale de Gasthuisberg doivent aussi être remerciés car ils se sont montrés très disponibles et ouverts aux discussions.

Je remercie enfin les différentes personnes qui ont accepté de lire ce mémoire, et dont les remarques ont contribué à clarifier le texte final.

Table des matières

Introduction

Les laboratoires de recherche médicale gaspillent une bonne partie de leurs ressources de personnel en tâches routinières et parfois même dangereuses. Il serait bon d'effectuer certains de ces travaux de façon automatisée. Des systèmes robotiques très flexibles et modulaires ont un avenir certain dans cet environnement (le lecteur souhaitant une introduction générale à la robotique pourra consulter [Br85]). Ces systèmes devant presque être configurés sur place, il est important qu'ils soient manipulables par le personnel de laboratoire lui-même. Ceci implique que leur logiciel de contrôle puisse être aisément adaptable et soit robuste aux erreurs humaines.

Les applications robotiques concernées comporteront presque toujours des tâches qui peuvent être exécutées en parallèle (car elles ne manipulent pas les mêmes objets et ne requièrent pas les mêmes appareils).

L'utilisation d'un langage offrant du parallélisme prend donc ici toute son importance car elle permet de prescrire de façon claire, dans le programme de l'application lui-même, les actions pouvant être exécutées en parallèle, et celles devant être synchronisées.

Un langage de programmation parallèle qui soit en plus logique offrirait la possibilité d'écrire le programme d'application comme une description logique du problème lui-même, et non comme une description procédurale d'une solution possible de ce problème. Nous constaterons cependant que cette vue est idéaliste : de tels programmes ne sont pas une description purement logique du problème car ils comportent des éléments de description procédurale.

L'objet de ce travail est d'étudier l'utilisation d'un langage logique parallèle dans le contexte général énoncé ci-dessus. Les langages logiques parallèles les plus connus sont Parlog [ClGr86] et Concurrent Prolog [Sh83]. Un nouveau langage basé sur Concurrent Prolog a été récemment décrit sous le nom de Guarded Horn Clauses (GHC) dans [Ue85a], [Ue85b]. Le langage plus particulièrement étudié ici sera le langage Guarded Horn Clauses.

Le travail réalisé est essentiellement composé de quatre parties correspondant chacune à un chapitre : une comparaison des trois langages logiques parallèles Parlog, Concurrent Prolog et GHC; le développement d'un interpréteur du langage GHC; un exemple d'application robotique en GHC; une méthodologie de conception d'applications robotiques en GHC.

Le *premier chapitre* comprend principalement une présentation synthétique et comparative des trois langages logiques parallèles Parlog (section I.3.), Concurrent Prolog (section I.4.) et GHC (section I.5.). Afin d'éliminer toute ambiguïté quant à la signification des termes employés dans cette présentation, une synthèse des concepts fondamentaux du parallélisme (section I.1.), ainsi qu'une présentation des concepts généraux liés aux langages logiques parallèles (section I.2.) sont données en préalable à l'analyse des trois langages. Ces trois langages sont, à la section I.6., comparés sur base d'une série de critères (puissance d'expression, degré de parallélisme effectif,. facilité de construction des programmes,...); c'est à cet endroit que le choix du langage étudié est justifié.

Afin de pouvoir utiliser effectivement le langage étudié, il était impératif de disposer d'un interpréteur ou d'un compilateur. Le *deuxième chapitre* concerne le développement d'un interpréteur du langage GHC. Le matériel disponible ne permettant que des exécutions séquentielles, des restrictions par rapport à une implémentation parallèle du langage GHC ont dû être apportées (section II.2.). Il s'est aussi avéré nécessaire de modifier légèrement la syntaxe de GHC (section II.3.). Les spécifications de l'interpréteur sont définies à la section II.4. et précèdent directement la description de la construction de l'interpréteur (section II.5.). Le procédé de construction utilisé est une construction par raffinements successifs : les différents sous-problèmes sont mis en évidence et à leur tour raffinés si leur degré de complexité s'avère trop élevé. Le code de l'interpréteur est donné à la section II.6. Une modification de la stratégie utilisée par l'interpréteur est proposée à la section II.7. Des comparaisons sont ensuite effectuées entre le langage GHC ainsi implémenté et deux langages qui lui sont fort proches : le langage Prolog séquentiel (section II.8.) et le langage Concurrent Prolog (section II.9.).

Grâce à l'interpréteur ainsi développé, il a été possible de rédiger et d'exécuter des applications robotiques en GHC. Le *troisième chapitre* propose une de ces applications. La description de celle-ci (section III.1.) correspond à un des exemples d'applications robotiques développés dans [Va85] : l'application des gels acryliques. La conception d'une telle application recouvre deux aspects importants. Le premier aspect concerne la gestion des appareils, faisant intervenir les problèmes de possibilités de pannes et de priorité entre les différents appareils (section III.3.). C'est aussi à ce niveau qu'est effectuée la description du lien entre l'application GHC et la couche objet proposée dans le cadre d'un environnement d'aide à la programmation de robot [Bet86] (section III.4.). Le second aspect est la synchronisation des tâches, qui doit être réalisée de façon à respecter les contraintes d'enchaînement décrivant l'application (section III.5.). Une brève discussion de l'adéquation du langage GHC à la robotique clôture ce chapitre (section III.6.).

Sur base des résultats obtenus pour l'application robotique ainsi développée, une proposition de méthodologie de conception d'applications robotiques en GHC

est présentée dans le *quatrième chapitre*. Cette méthodologie a pour but de transformer systématiquement une application séquentielle décrite en Prolog en une application parallèle décrite en GHC. Le programme GHC ainsi obtenu se devant d'être correct, le concept de validité d'un programme est défini à la section IV.1. Cette définition et les caractéristiques des descriptions séquentielles de tâches d'une application robotique sont les bases sur lesquelles la méthodologie repose. La description de la transformation à effectuer, ainsi que l'étude de sa validité sont abordées à la section IV.2. La méthodologie ainsi décrite est ensuite appliquée à un exemple complet correspondant à l'application développée dans le troisième chapitre (section IV.3.). L'existence de certaines restrictions dans le contexte d'applicabilité de cette méthodologie nous a amenés à en suggérer des extensions (section IV.4.).

Deux résultats fondamentaux ont été atteints par le biais de ce travail.
Le premier est le développement de l'interpréteur de GHC : à notre connaissance, aucun interpréteur n'avait jusqu'à présent été développé.

Le second concerne l'étude de l'utilisation du langage logique parallèle GHC dans le cadre d'applications robotiques, et plus particulièrement la proposition de méthodologie de programmation sous-jacente. GHC étant un langage plus récent que Parlog et Concurrent Prolog, les recherches effectuées le concernant sont beaucoup moins nombreuses que celles déjà réalisées pour les deux autres langages. L'évaluation des avantages et inconvénients de GHC est donc loin d'être terminée. Dans un tel contexte, l'étude réalisée dans ce travail peut être considérée comme une contribution originale à l'étude du langage GHC.

Chapitre I

Les langages de programmation logique parallèle

I.1. Concepts fondamentaux du parallélisme

I.1.1. Introduction

Comme l'a suggéré Luc De Vos [Dv85], parler de parallélisme en programmation signifie considérer des parties de programmes bien définies (appartenant à plusieurs programmes différents, ou issues du même programme) comme étant logiquement indépendantes. Le parallélisme est donc virtuel, au sens où les exécutions de ces parties peuvent être effectuées dans un ordre tout à fait arbitraire car elles sont logiquement indépendantes. Bien qu'un programme parallèle puisse être exécuté sur un processeur unique, cela peut évidemment donner lieu à un parallélisme réel et physique (si on dispose du matériel adéquat).

Cette mise au point étant faite, nous pouvons passer à la définition des principaux concepts liés au parallélisme. Ces définitions sont basées principalement sur les articles [Di71] et [Ha73].

I.1.2. Processus

La notion de processus est *dynamique*. Un processus est l'exécution d'une suite d'opérations (que ce soient des actions ou des évaluations) avec compétition pour des ressources. Dans le cadre de la programmation logique, une ressource peut être l'accès à une variable et un processus l'essai d'établissement d'un prédicat.

Des processus sont appellés *concurrents* si leurs exécutions se chevauchent dans le temps, c'est-à-dire si la première opération d'un processus débute avant que la dernière opération d'un autre processus ne soit terminée.

I.1.3. Interférence

Une *interférence* est une interaction entre plusieurs processus qui peut conduire à un comportement erroné du système. Il faut éviter de telles situations, et c'est pour cela que des contraintes d'exclusion mutuelle et/ou de

synchronisation sont imposées au niveau des interactions entre processus concurrents.

De là, on voit apparaître deux types de processus : des processus disjoints, qui n'ont aucune interférence entre eux, et des processus interférant (ou coopérant).

I.1.4. Section critique, exclusion mutuelle et équitabilité

La notion de section critique est *statique*. Une section critique est une suite atomique d'opérations. Cette suite d'opérations doit donc être exécutée de façon indivisible et ininterruptible. Cette suite correspond à une portion statique de programme que le processus exécute. C'est pour éviter des interférences que des parties de programme peuvent être définies comme sections critiques, induisant ainsi une exclusion mutuelle.

Les sections critiques doivent vérifier trois propriétés :
- *exclusion mutuelle* : au maximum un processus à la fois peut être dans la section critique.
- *terminaison* : un processus sortira toujours d'une section critique dans un délai fini.
- *équitabilité* : un processus pourra toujours entrer dans une section critique dans un délai fini.

I.1.5. Synchronisation

Pour éviter les interférences, les processus sont également amenés à se synchroniser entre eux. La synchronisation recouvre deux concepts : l'attente et la signalisation.

- *Attente (blocage) :*

Si la condition qui régit l'avancement d'un processus n'est pas satisfaite, le processus sera bloqué en attendant la survenance de certains événements susceptibles de rendre cette condition valide.
Ainsi, dans un schéma simple d'exclusion mutuelle, lorsqu'un processus A se trouve dans une section critique, il est impossible à un autre processus B d'y entrer. Le processus B sera bloqué et attendra la survenance d'un événement rendant possible son entrée dans la section critique.

- *Signalisation (réveil) :*

Quand un processus rend une condition de synchronisation vraie, il doit produire un signal pour rendre publique la survenance de cet événement. Ce signal autorisera un des processus en attente (choisi arbitrairement) à continuer son exécution.

Dans l'exemple ci-dessus, quand un processus sort d'une section critique, l'événement est signalé et on réveille au hasard un des processus en attente d'entrer dans la section critique concernée. Le processus peut alors continuer son exécution.

I.1.6. Mécanisme de communication

La synchronisation entre processus suppose l'existence d'un mécanisme de communication pour signaler la survenance d'événements.

Il y a principalement deux façons d'implémenter ce mécanisme de communication :

- par *mise-à-jour de variables partagées* qui sont des variables communes à plusieurs processus (cfr. les sémaphores, régions critiques ou moniteurs en programmation impérative parallèle, ou Parlog, Concurrent Prolog et GHC en programmation logique parallèle). Cette technique est la plus classique des deux et la plus répandue.
- par *échange de messages* via un canal de communication (cfr. CSP [Ho78] en programmation impérative, ou DELTA-PROLOG en programmation logique). Cette technique est beaucoup plus récente.

I.1.7. Non-déterminisme

Le non-déterminisme peut être défini comme étant l'impossibilité à partir d'un état du système de savoir quel en sera l'état suivant parmi un ensemble d'états possibles. Ainsi, le choix de l'action correspondante à exécuter est effectué de façon arbitraire.

Toujours dans l'exemple ci-dessus, en fin de section critique on signale la survenance d'un événement mais on ne sait pas quel processus sera réveillé parmi les plusieurs possibles, bloqués à l'entrée de la section critique.

Les problèmes de nature non-déterministe sont une des principales sources d'exploitation du parallélisme. En effet, des programmes non-déterministes donnent souvent lieu à une possibilité d'exécution parallèle [Dv85].

Cette notion de non-déterminisme sera approfondie à la section I.2.7., dans le cadre de la définition d'une clause gardée. Le lien existant entre le non-déterminisme et une clause gardée est dû au fait que la sémantique procédurale d'une telle clause est en grande partie inspirée de la construction des commandes gardées de Dijkstra [Di75].

I.1.8. Problèmes de synchronisation

La synchronisation entre processus peut entraîner des problèmes du même type que les problèmes de terminaison en programmation séquentielle. On peut

envisager essentiellement deux types de tels problèmes : l'interblocage et le blocage individuel permanent.

L'*interblocage* peut être défini comme une situation dans laquelle ***aucun*** des processus concurrents ne peut poursuivre son exécution parce qu'ils sont tous en attente.
Dans le cadre d'un partage de ressources, l'interblocage apparaît par exemple lorsque deux processus attendent chacun une ressource déjà allouée à l'autre [Co81], ces ressources étant non préemptibles.

Le *blocage individuel permanent* (aussi connu sous le nom de famine) consiste en l'attente indéfinie d'***un*** processus, par exemple à cause de l'application d'une règle de priorité pour l'allocation de ressources [Co81], ou à cause de coalitions de processus contre le processus "affamé" [Di71].

1.2. Concepts généraux liés aux langages logiques parallèles

Les langages logiques parallèles sont conçus pour prescrire l'exécution parallèle de programmes logiques. La plupart de ces langages sont basés sur le Prolog séquentiel et ont la même (ou presque) lecture déclarative que Prolog, ils diffèrent par la sémantique procédurale.

Seuls seront évoqués ici les concepts utilisés dans la description de la syntaxe et surtout de la sémantique procédurale des trois langages logiques parallèles Parlog, Concurrent Prolog et GHC.
Le nombre de concepts nécessaires à la description des trois sémantiques procédurales est un indice de leur complexité. Le problème de cette complexité sera abordé dans le cadre de la comparaison des trois langages, à la section I.6.4.

Une certaine familiarité avec la programmation logique et le langage Prolog séquentiel est supposée acquise. Le lecteur pourra consulter à cet effet [Hog84], [Ll84] et [ClMe81].

I.2.1. Termes et prédicats

• Les *termes* sont utilisés pour désigner des objets. On peut les classer en deux catégories : termes simples et termes composés.

◇ Les *termes simples* comprennent les constantes et les variables. Une *constante* peut être soit un atome, soit une constante numérique. Une *variable* dénote une classe d'objets.

◇ Les *termes composés* (ou *structures*) se présentent sous la forme **F(t1,...,tn)** où chaque argument **ti** est un terme et où **F** est un foncteur.

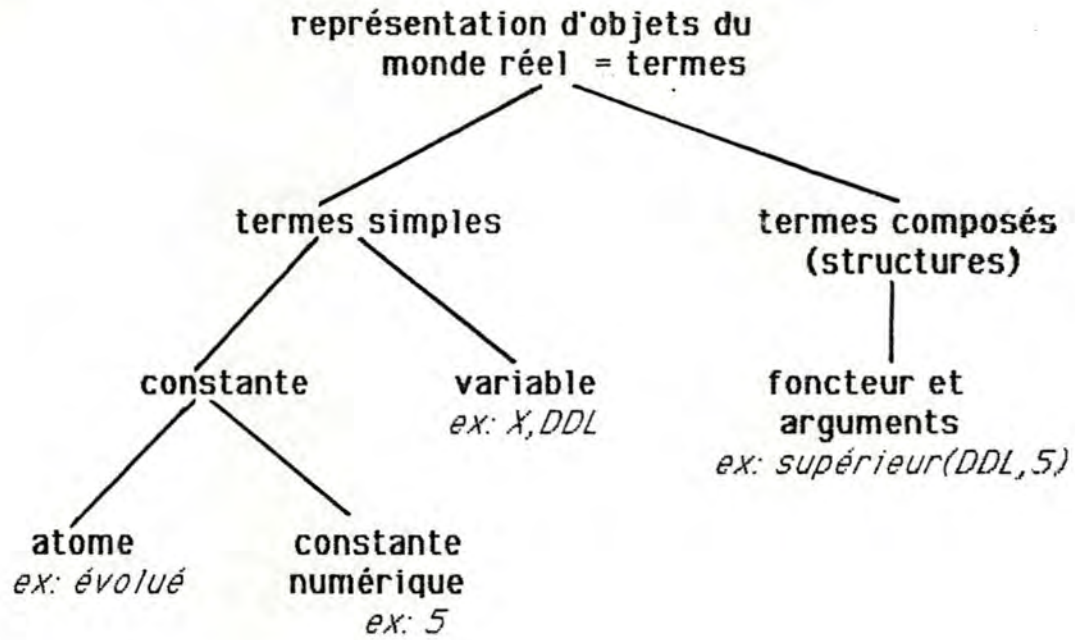
Dans la suite, l'expression *terme non variable* sera utilisée pour désigner une constante ou une structure dont tous les arguments sont des constantes. On désignera par *terme variable* une variable non instanciée ou une structure dont au moins un des arguments est une variable non instanciée.

Illustrons les éléments de cette terminologie au moyen d'un exemple simple :

" Un robot appartient à la catégorie 'évolué' s'il est un automate et s'il possède plus de 5 degrés de liberté."

```
catégorie_robot(X,évolué) :- automate(X),
                             degrés_de_liberté(X,DDL),
                             supérieur(DDL,5).
```

Résumons les concepts introduits sous forme d'arbre :



• Les *prédicats* (ou *formules*) peuvent être établis ou échouer. La forme syntaxique d'un prédicat comprend un nom de prédicat (P) et un ensemble d'arguments qui sont des termes t1,...,tn , soit P(t1,...,tn).
Un prédicat peut ne pas avoir d'arguments (n = 0).

Exemples de prédicats :

automate(X)
degrés_de_liberté(X,DDL)
supérieur(DDL,5)

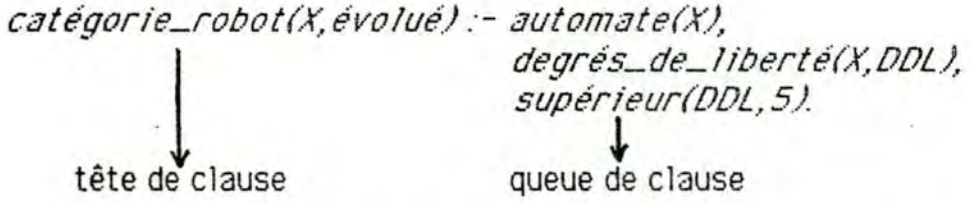
I.2.2. Procédure

• Une *clause* (ou *règle*) est une affirmation générale à propos d'objets et de leurs relations. La forme syntaxique d'une clause peut être caractérisée de la façon suivante :

< tête > :- < queue >

La tête est un prédicat et la queue (ou corps) est formée d'une suite éventuellement vide de prédicats. Les prédicats apparaissant dans la queue de la clause sont aussi appellés les sous-objectifs de la clause.

Exemple de clause :



• Une *procédure* est un ensemble fini de clauses dont les têtes ont le même foncteur et la même arité (le même nombre d'arguments). Les clauses sont séparées par des points.

Exemple de procédure :

```
catégorie_robot(X,primaire) :- automate(X),
                               degrés_de_liberté(X,1).
catégorie_robot(X,évolué) :- automate(X),
                             degrés_de_liberté(X,DDL),
                             supérieur(DDL,5).
```

• Un *programme* est un ensemble fini de procédures.

I.2.3. Unification

L'*unification* de deux formules est le processus qui permet de déterminer s'il existe un jeu de substitutions de variables rendant les deux formules formellement identiques. Ce mécanisme de substitution peut entraîner l'instanciation d'une variable de la première formule à la suite de symboles formant l'argument correspondant dans la seconde formule.

Quand deux formules à unifier sont deux variables X et Y, le résultat de cette unification est que toute manipulation ultérieure de l'une se répercutera sur l'autre, et vice versa.

L'*unification restreinte*, ou *pattern matching*, est une équivalence formelle, sans substitution. Les deux formules à unifier sont identiques symbole par symbole, aucune instanciation de variable n'aura donc lieu.

I.2.4. Résolution et appel

La *résolution* d'une conjonction d'objectifs est effectuée par réductions successives des objectifs. Une étape de résolution consiste à choisir un objectif de la conjonction et une clause dont la tête s'unifie avec l'objectif considéré. On réduit alors la conjonction restant à résoudre en y remplaçant l'objectif considéré par le corps (éventuellement vide) de la clause qui a été utilisée.

Le processus de résolution se termine lorsque la conjonction d'objectifs restant à résoudre est vide (succès) ou lorsqu'aucune des clauses du programme ne peut être utilisée pour réduire les objectifs restants (échec).

L'*appel* est l'objectif courant à établir. L'appel est libellé sous la forme d'un prédicat dont les arguments sont appellés les paramètres ou variables de l'appel. Pour résoudre l'appel, on recherche une clause dont la tête s'unifie avec l'objectif courant à établir.

Soit la clause : **q(X) :- p(X), p(X).**
Pour la résolution de la conjonction d'objectifs du corps de la clause ci-dessus, on considère successivement deux objectifs courants à établir (qui correspondent tous deux au même objectif p(X)). On se trouve donc en présence de deux appels à résoudre.

I.2.5. Exportation de liaison

Une *exportation de liaison* est l'exportation à partir d'une tête de clause vers un appel d'une liaison entre l'appel et la tête de clause. La clause considérée est une des clauses de la procédure correspondant à l'appel, elle est utilisée pour la résolution de l'appel.

Une telle exportation a lieu lorsque le résultat de l'unification de l'appel considéré et de la tête de clause est l'instanciation d'une variable apparaissant dans l'appel à la suite de symboles formant l'argument correspondant dans la tête de clause.

I.2.6. Processus

Il est intéressant, afin de bien faire le lien avec la section I.1., de définir le concept de processus dans le cadre d'un langage logique parallèle.

Un processus est une évaluation d'objectif, un essai d'établir cet objectif. Dans le cas d'une conjonction d'objectifs, les processus correspondant aux objectifs communiquent s'ils ont parmi les arguments de ceux-ci des variables communes.

I.2.7. Garde d'une clause

• Une *clause gardée* peut, en toute généralité, être caractérisée par la syntaxe suivante :

$$H :- G1, G2, \ldots, Gm \ / \ B1, B2, \ldots, Bn. \qquad \text{avec } m,n \geq 0$$

H est la tête de la clause,
les Gi forment la garde,
les Bi forment le corps,
H, Bi et Gi sont des formules pouvant contenir des variables comme arguments.

L'opérateur " | " sépare la garde du corps de toute clause gardée. Des différences de terminologie existent entre les langages logiques parallèles en ce qui concerne son nom. Il ne porte pas de nom spécial en Parlog, il est appellé "opérateur de commit" en Concurrent Prolog, et "opérateur de trust" en GHC. Dans la suite de ce chapitre consacré aux concepts généraux, nous l'appellerons "opérateur de commit".

Bien que cela ne soit pas essentiel, on peut signaler que des variations sont aussi présentes dans les cas où m ou n = 0. En ce qui concerne Parlog et Concurrent Prolog, l'opérateur de commit peut être omis si m=0. Pour GHC, le prédicat vide "true" est utilisé pour noter une garde ou un corps vide (m ou n = 0), l'opérateur " | " ne peut donc jamais être omis dans une clause GHC.

• La *sémantique déclarative* d'une clause gardée est analogue à celle d'une clause Prolog. En effet, les virgules séparant les différents sous-objectifs Gi et Bi sont en fait des 'et', l'opérateur ' | ' étant aussi lu comme un 'et'. La déclaration d'une clause gardée peut donc être interprétée de la façon suivante : pour tout jeu de valeurs des variables de la clause gardée, H est vrai si tous les Gi et tous les Bi sont vrais.

La sémantique déclarative d'une clause gardée étant la même pour les trois langages étudiés, elle ne sera pas rappellée dans les descriptions propres à chaque langage.

• La *sémantique procédurale* d'une clause gardée est en grande partie inspirée de la construction des commandes gardées de Dijkstra [Di75].

Rappellons que la structure de base des commandes gardées de Dijkstra consiste en la construction :

garde ---> liste gardée

La garde est formée d'une expression booléenne et la liste gardée, d'une liste d'opérations. Cette structure de base peut être utilisée pour construire une commande alternative ou répétitive.

commande alternative :

```
if  G1 ---> GL1
 □  G2 ---> GL2
      ...
 □  Gn ---> GLn
fi
```

commande répétitive :

```
do  G1 ---> GL1
 □  G2 ---> GL2
      ...
 □  Gn ---> GLn
od
```

La *sémantique de la commande alternative* est la suivante :

◇ si dans l'état initial aucune des gardes Gi n'est vraie, le programme va avorter;

◇ sinon , une liste gardée GLj dont la garde Gj est vraie sera sélectionnée de façon arbitraire pour l'exécution.

L'ordre dans lequel les commandes gardées apparaissent dans le texte est non significatif car il y a non-déterminisme quant au choix de la liste gardée GLj.

La *sémantique de la commande répétitive* diffère de celle de la commande alternative. La différence se situe au niveau du cas où aucune garde

n'est vraie. Ce cas ne conduit pas à l'avortement du programme, mais bien à sa terminaison naturelle. Si, dans l'état initial ou à la fin de l'exécution d'une liste gardée, une ou plusieurs gardes sont vraies, on sélectionnera à nouveau de façon arbitraire une liste gardée GLj parmi celles dont la garde est vraie. L'exécution de cette commande prendra donc fin seulement quand toutes les gardes seront fausses.

• Deux définitions permettront de mieux comprendre le rôle logique d'une garde dans une clause :

1. L'*évaluation d'une garde* est la résolution de l'objectif formé par les différents prédicats apparaissant dans la garde de la clause considérée. La résolution utilise les clauses du programme afin de réduire cet objectif à la clause vide.

2. Une *garde est satisfaite* si et seulement si le mécanisme de résolution arrive à la clause vide pour la réduction de la conjonction d'objectifs associée à la garde.

Soit la procédure avec clauses gardées :

```
P :- G1 / B1.
P :- G2 / B2.
P :- G3 / B3.
```

et l'appel : *?- P.*

La sémantique de cet appel de procédure à clauses gardées sera dès lors la suivante :

Si dans l'état initial aucune des gardes Gi n'est satisfaite, l'appel correspondant au prédicat P va échouer;

sinon, un corps de clause Bj dont la garde Gj est satisfaite sera sélectionné arbitrairement. C'est Bj qui sera utilisé pour la résolution de l'appel.

Tout comme pour les commandes gardées de Dijkstra, l'ordre dans lequel les clauses gardées apparaissent dans la procédure est non significatif, puisqu'il y a non-déterminisme quant au choix de la clause j parmi toutes les clauses dont la garde est satisfaite.

Les évaluations des gardes des clauses d'une procédure sont effectuées en parallèle.

Une restriction doit toutefois être faite dans le cadre de Parlog. En effet, comme cela sera expliqué à la section I.3.3., ce langage permet d'accorder une importance à l'ordre des clauses d'une procédure en offrant la possibilité de prescrire une recherche séquentielle pour le choix de la clause j.

I.2.8. Opérateur de "commit" d'une clause

Etant donnée la clause *H :- G1, ... , Gm / B1, ... , Bn*, l'opérateur " / " est un prédicat au même titre que les autres prédicats Gi ou Bi.

Ce prédicat est exécuté après que la garde soit satisfaite, c'est-à-dire lorsque le système *G1, ... ,Gm* est résolu.

L'exécution de l'opérateur de "commit" pour une clause i d'une procédure a pour effet de :
1) stopper les évaluations des gardes des autres clauses de la procédure : seule la clause pour laquelle le "commit" a été exécuté sera considérée par la suite,
2) réduire l'appel au corps de la clause i.

Soit la procédure :	*H :- G1 / B1.*	(1)
	H :- G2 / B2,B3.	(2)
et l'appel :	*?- H.*	(3)

Supposons que la garde de la clause (2) soit satisfaite et que celle de la clause (1) soit toujours en cours d'évaluation. Le "commit" est exécuté pour la clause (2). L'évaluation de la garde G1 de la clause (1) est immédiatement stoppée. L'appel (3) pour lequel la résolution est exécutée sera alors réduit au corps de la clause (2) : *B2,B3.*

L'opérateur de "commit" réussit toujours de façon inconditionnelle. Les mécanismes implémentés afin de s'assurer qu'il n'y a qu'une clause par procédure qui passe le "commit" seront abordés dans le cadre de chacun des trois langages étudiés.

I.2.9. Corps d'une clause

L'exécution du corps d'une clause est la résolution de l'objectif formé par les différents prédicats apparaissant dans le corps. La résolution utilise les clauses du programme afin d'essayer de réduire cet objectif à la clause vide.

Normalement, l'exécution du corps d'une clause ne peut commencer que si cette clause a passé le "commit". Nous verrons que le langage GHC est une exception à cette règle puisqu'il permet que les exécutions du corps et de la garde d'une même clause soient réalisées en parallèle, pour autant que certaines règles soient respectées.

L'exécution du corps d'une clause échoue si le mécanisme de résolution ne parvient pas à la clause vide. Dans ce cas, il n'y a pas de rétroparcours sur le choix de cette clause parmi des différentes clauses candidates de la procédure.

La clause qui effectue le "commit" étant unique, l'échec de la résolution de son corps implique l'échec de la résolution de l'appel de départ.

L'*absence de rétroparcours sur le choix arbitraire d'une clause* pour la résolution correspond au "committed choice non determinism" des commandes gardées de Dijkstra [Di75] et de CSP de Hoare [Ho78].

Cette contrainte implique l'obligation de programmer dans un style fort différent de celui de la programmation logique classique, puisque le choix de la clause dont l'évaluation de la garde est terminée est tout à fait arbitraire. Ce choix est irréversible dès qu'il est confirmé par le "commit" pour la clause considérée. En construisant le programme, on doit dès lors construire les gardes et les termes des têtes de clauses de façon à s'assurer que, si la tête d'une clause s'unifie avec l'appel et que sa garde est satisfaite, alors une solution peut être trouvée en utilisant cette clause [Ko79a].

En effet, si tel n'est pas le cas (s'il n'est pas possible de trouver une solution pour l'appel en utilisant cette clause), il est certain qu'aucune solution ne pourra être trouvée en utilisant les autres clauses [ClGr86], puisque l'utilisation de celles-ci est éliminée par le passage du "commit" par la clause sélectionnée.

I.2.10. Etats d'une clause

Trois états significatifs d'une clause sont à envisager : candidate, suspendue et non-candidate. Ils sont tous trois liés à l'évaluation de la clause dans le cadre de la résolution d'un objectif donné.

Une *clause candidate* pour la résolution d'un objectif est une clause dont la tête s'unifie avec l'objectif et dont l'évaluation de la garde s'est terminée avec succès.

Une *clause suspendue* est une clause dont l'unification de la tête avec l'objectif ou la résolution d'un prédicat de la garde a du être suspendue par besoin de synchronisation. Ce besoin de synchronisation peut être dû à un processus qui voudrait exporter de façon illicite une liaison à l'appel(en GHC) ou à un processus voulant utiliser des variables de type 'input' qui ne sont pas encore instanciées (en Concurrent Prolog ou en Parlog). Notons qu'une variable est de type 'input' pour un processus s'il doit disposer d'une valeur pour celle-ci avant de pouvoir la référencer. Si toutes les clauses de la procédure correspondant à l'appel dont la résolution est en cours sont suspendues, on dit que l'appel est suspendu. L'expression 'clause suspendue' est un petit abus de langage au sens où c'est l'évaluation (l'unification de la tête ou l'évaluation de la garde) qui est suspendue, et non la clause elle-même.

Une *clause non-candidate* est une clause pour laquelle soit l'unification de la tête avec l'objectif à résoudre a échoué, soit la réduction de la garde à la clause vide n'est pas possible. Il faut insister sur le fait que ces échecs sont

définitifs en ce sens qu'ils ne dépendent pas d'une suspension. Si toutes les clauses disponibles pour la résolution d'un appel donné sont non-candidates, l'appel échoue.

L'*évaluation d'une clause* comporte l'unification de sa tête avec l'objectif courant à établir (l'appel) et l'évaluation de sa garde. Tant qu'au moins une de ces deux actions n'est pas terminée, la clause est en évaluation. Les états 'candidate' et 'non-candidate' sont définis en fin d'évaluation d'une clause, tandis que 'suspendue' s'applique à une clause dont l'évaluation est toujours en cours.

I.2.11. Parallélisme-ET et parallélisme-OU

Deux formes de parallélisme peuvent être distinguées : parallélisme-ET et parallélisme-OU.

L'évaluation gauche - droite des sous-objectifs dans une clause Prolog peut être remplacée par une résolution de ceux-ci en parallèle; ceci s'appelle le *parallélisme-ET*.

L'ordre séquentiel, en Prolog, dans lequel les clauses alternatives sont essayées pour résoudre un objectif peut être remplacé ou augmenté par la possibilité d'essayer toutes les alternatives en parallèle; ceci est le *parallélisme-OU*.

Tout comme existent deux formes de parallélisme, nous pouvons définir deux formes de non-déterminisme (cfr [Ko79]).

Non-déterminisme-ET: quand plusieurs sous-objectifs apparaissent dans le corps d'une clause, l'ordre dans lequel ils seront résolus n'est pas déterminé-ET.

Non-déterminisme-OU: quand plusieurs clauses ont une tête qui s'unifie avec un appel donné, la stratégie au moyen de laquelle les clauses alternatives seront essayées n'est pas déterminée-OU. Ce non-déterminisme est une caractéristique essentielle du langage de commandes gardées de Dijkstra [Di75].

Certains problèmes spécifiques sont liés à l'existence des parallélisme-ET et parallélisme-OU dans les langages de programmation.

Les problèmes principaux apparaissent avec le parallélisme-ET, quand deux ou plusieurs sous-objectifs contiennent des termes qui partagent une variable non instanciée. Ce problème correspond à la notion d'interférence définie à la section I.1.3. Seulement un de ces sous-objectifs devrait pouvoir faire l'instanciation, car il y a risque d'inconsistance.

Les langages qui permettent un parallélisme-ET limité forcent les sous-objectifs qui partagent des variables à être exécutés de façon strictement séquentielle, mais permettent aux sous-objectifs sans variables partagées d'être exécutés en parallèle. Cette solution correspond au principe d'exclusion mutuelle sur section critique énoncé à la section I.1.4.

Des problèmes existent aussi avec le parallélisme-OU. Les différentes clauses qui sont essayées en parallèle pour la réduction d'un appel tel que A(X,Y) pourraient, si aucune précaution n'est prise, exporter vers l'appel des liaisons en instanciant une des deux variables (X et/ou Y) de l'appel. Si tel est le cas, on a un risque d'inconsistance car plusieurs processus (correspondant chacun à une des clauses essayées en même temps) travaillent sur un même ensemble de variables partagées. Le risque d'interférence existe donc aussi au niveau du parallélisme-OU.

Les langages qui permettent un parallélisme-OU limité interdisent aux différentes clauses essayées en parallèle pour un même appel d'exporter vers celui-ci des liaisons avant que le choix de la clause qui servira à la résolution de l'appel ne soit définitif. A partir du moment où la clause choisie est déterminée, les évaluations des autres clauses sont arrêtées. Cette clause sélectionnée peut donc exporter des liaisons à l'appel, sans aucun problème d'interférence puisqu'elle est unique. Contrairement à ce qui se passe pour le parallélisme-ET, nous n'avons pas vraiment un mécanisme d'exclusion mutuelle sur section critique puisque les interférences sont évitées en supprimant l'existence-même de la section critique. En effet, interdire aux processus exécutés en parallèle de modifier la valeur des variables qu'ils manipulent est une limitation du parallélisme qui supprime tout risque d'interférence. Comme les variables communes ne peuvent être consultées qu'en lecture, les processus concernés peuvent être considérés comme disjoints et dès lors assimilés à des processus séquentiels n'ayant pas de relation entre eux (cfr. section I.1.3.). Nous verrons comment cette interdiction de manipulation simultanée des mêmes variables est concrètement réalisée pour les langages étudiés dans les trois sections qui suivent.

I.3. Le langage PARLOG

Les caractéristiques de Parlog présentées ici sont extraites de [ClGr86].

I.3.1. Remarque préalable

Nous avons vu à la section I.2.9. que l'absence de rétroparcours sur le choix de la clause d'une procédure utilisée pour résoudre un appel entraîne l'obtention d'une solution unique. Toutefois, il existe des applications en programmation logique où toutes les solutions possibles sont demandées (essentiellement dans le contexte des bases de données).

Pour permettre ce type d'applications, les procédures de Parlog sont divisées en deux types : *procédures à une solution* (single-solution) et *procédures à solutions multiples* (all-solutions). Les procédures à une solution sont définies par des programmes avec clauses gardées, tandis que les procédures à solutions multiples sont des séquences de clauses Prolog.

Pour une conjonction d'appels à des procédures à solutions multiples, des constructeurs d'ensemble (set constructors) produisent une liste contenant toutes les solutions (ou du moins plusieurs). Les procédures de ce type permettent donc, en offrant la possibilité d'obtenir plusieurs solutions, de remédier à l'absence de rétroparcours telle qu'elle a été spécifiée à la section I.2.9.

Tout ce qui concerne les procédures à solutions multiples forme une partie de Parlog qui lui est propre, aucune construction de ce genre n'étant présente dans la description de Concurrent Prolog ou de GHC. Aucune comparaison avec les deux autres langages n'est donc possible. Comme, de plus, l'utilisation de cette caractéristique se situe principalement dans le cadre de l'exploitation de bases de données et que ce type d'application n'est pas spécialement concerné par le présent travail, *seule sera considérée dans la suite la composante de Parlog consacrée aux procédures à une solution.*

I.3.2. Syntaxe

• Les *variables* sont des noms commençant par une lettre minuscule, tout comme les prédicats. Les atomes commencent par une lettre majuscule. L'opérateur de "commit" est représenté par le symbole ' : '. Un exemple de clause Parlog correspondrait à la forme suivante :

h(TRUE,x,y) <-- < garde > : < corps >.

Néanmoins, il serait plus clair d'utiliser la même syntaxe pour présenter les trois langages. On utilisera donc la syntaxe qui est commune à Concurrent Prolog et GHC. Elle peut être caractérisée de la façon suivante : les variables commencent par des lettres majuscules, les atomes et prédicats par des lettres minuscules, et l'opérateur de "commit" est représenté par le symbole ' | '.
La clause Parlog donnée ci-dessus sera donc représentée sous la forme :

h(true,X,Y) :- < garde > | < corps >.

• La description d'une procédure comprend une déclaration de mode unique et une séquence de clauses gardées (l'expression "séquence" est employée, plutôt qu' "ensemble", car l'ordre entre les clauses est significatif) (cfr point a de la section I.3.3.).

• La *déclaration de mode* d'une procédure **r** a la forme suivante :

mode r(m1, ..., mk).

Chaque mi de la déclaration de mode vaut '**?**' ou ' **^** ' et est optionnellement précédé d'un identifiant sans signification sémantique (utilisé seulement comme commentaire).

Considérons l'appel **r(A)** et la procédure : **mode r(?).**
r(T) :- ...

Un argument annoté avec un '**?**' dans une déclaration de mode est dit de *type input*. Cela signifie que l'unification d'un terme non variable **T** apparaissant à la position de cet argument dans la tête d'une clause de la procédure **r** avec l'argument correspondant **A** dans l'appel réussira seulement si **A** est un terme non variable qui est unifiable à **T**.

Exemple : appels : **r([true|X]).** (1)
r(X). (2)
procédure : **mode r(argument1 ?).**
r([A|B]) :- ...

Pour l'appel (1), l'unification réussira car l'argument de l'appel est un terme non variable et il existe une substitution possible : A --> true, B --> X.

Pour l'appel (2), l'unification ne réussira pas car l'argument de l'appel est une variable. Nous verrons plus loin ce que cet échec implique au niveau de la sémantique procédurale de Parlog.

Considérons l'appel **r(V)** et la procédure : **mode r(^).**
r(T) :- ...

Une annotation ' **^** ' dans une déclaration de mode désigne un argument de *type output*. Cela signifie que l'unification d'un terme non variable **T** à la position de cet argument dans la tête d'une clause de la procédure **r** avec l'argument correspondant A dans l'appel ne réussira que si **A** est une variable non instanciée.

Exemple : appels : **r(X).** (1)
r(true). (2)
procédure : **mode r(argument1 ^).**
r(Y) :- .....

Pour l'appel (1), l'unification réussira car l'argument de l'appel est une variable.

L'appel (2) entraînera un échec de l'unification puisque son argument est un atome. Les conséquences d'un tel échec seront abordées dans la section concernant la sémantique procédurale.

• Une *clause gardée* est une clause de la forme suivante :

r(t1, ..., tk) :- <conditions de la garde> | <conditions du corps>.

La barre ' **|** ' sépare la garde du corps et est omise si la garde ne comporte aucune condition (est vide). **t1, ..., tk** sont les termes arguments de la tête de clause (cfr section I.2.7.). Les appels apparaissant dans les conditions de la garde et du corps correspondent à des procédures à une solution.

I.3.3. Sémantique procédurale

a) Ordre d'essai des clauses

Les clauses d'une procédure sont séparées les unes des autres par les opérateurs '.' ou ';'. Ces opérateurs définissent l'ordre dans lequel les clauses sont essayées pour trouver une clause candidate pour la résolution d'un appel donné.

Les clauses séparées par un '.' doivent être essayées en parallèle-OU, tandis qu'un ';' indique que les clauses suivantes ne doivent être essayées que si toutes les clauses précédentes sont non-candidates. Notons que si une des clauses précédant le ';' est suspendue, la recherche d'une clause candidate ne dépassera pas ce ';' (une clause suspendue n'est ni candidate, ni non-candidate, cfr section I.2.10.).

La recherche parallèle-OU d'une clause candidate peut donc être remplacée par une recherche séquentielle en utilisant des opérateurs ';' entre toutes les clauses.

b) Unification de tête de clause

Comme nous l'avons vu à la section I.3.2., la déclaration de mode d'une procédure exprime les contraintes sur l'unification d'un appel et d'une tête de clause. Une telle unification peut ne pas satisfaire aux contraintes de mode.

• Dans le cas d'un argument input, si le *mode input* n'est pas respecté car l'argument correspondant de l'appel est une variable, l'unification est simplement suspendue. Reprenons l'exemple vu à la section I.3.2. :

appel : **r(X).**
procédure : **mode r(?).**
r([A|B]) :- ....

L'unification de **r(X)** avec **r([A|B])** sera suspendue car **X** est une variable. La clause est donc dans l'état 'suspendue'. Cette variable **X** devra être instanciée par un autre processus (par unification avec un autre appel). Dès que **X** a reçu une valeur, on reprend l'évaluation de la clause.

Si le mode input n'est pas respecté car l'argument de l'appel et de la tête de clause (tous deux non variables) ne sont pas unifiables, l'unification de l'appel et de la tête de clause se termine sur un échec.

Si la valeur attribuée à **X** rend l'unification avec la tête de clause **r([A|B])** impossible, l'essai d'utiliser cette clause se terminera sur un échec et la clause devient non-candidate.

• Si c'est un *mode output* qui n'est pas respecté car l'argument de l'appel n'est pas une variable non instanciée, le résultat de l'unification est une erreur. L'exemple suivant est une illustration de ce cas :

appel : **r(true).**
procédure : **mode r(^).**
r(Y) :- ...

Pour vérifier la contrainte liée à la déclaration d'arguments de type output, l'argument correspondant dans l'appel doit être une variable non instanciée. Or, cet argument est ici un atome (true). Le non-respect du mode entraîne une erreur dans l'exécution de l'unification.

Comme nous l'avons vu, l'unification est une activité qui sera parfois suspendue, pour ne reprendre que plus tard. Elle peut donc être considérée comme une activité continue qui ne se termine que sur un succès ou un échec définitif.

c) Evaluation de la garde

L'évaluation de la garde peut être réalisée en parallèle avec l'unification de l'appel et de la tête de clause.

Si la garde est vide, la clause est candidate dès que l'unification de sa tête avec l'appel a réussi.

Les sous-objectifs de la garde forment une conjonction d'appels à des procédures Parlog. Cette conjonction est construite en utilisant les opérateurs de conjonction ',' ou '&'. La virgule ',' est la conjonction parallèle, tandis que le '&' est la conjonction séquentielle. Ainsi, (C1 , C2) signale que C1 et C2 doivent être évalués en parallèle-ET, alors que (C1 & C2) indique que C2 ne doit être évalué que si l'évaluation de C1 se termine avec succès. On peut donc séquentialiser l'évaluation des sous-objectifs d'une garde.

Considérons le cas où l'évaluation des sous-objectifs de la garde se fait en parallèle-ET. Les variables partagées par cet ensemble de sous-objectifs servent

de support de communication entre les processus correspondant aux évaluations de chacun de ces sous-objectifs (cfr section I.1.6.). En utilisant les déclarations de mode pour contraindre l'unification d'un appel et d'une tête de clause, on peut limiter à un seul le nombre de processus capables d'instancier une variable partagée. Le sous-objectif évalué par ce processus est un appel à une procédure pour laquelle l'argument correspondant à la variable partagée est de type output. Ce processus est appellé le producteur pour la variable. Tous les autres processus correspondant aux procédures pour lesquelles la variable partagée est un argument de type input sont appellés les consommateurs de la variable. Les consommateurs, qui ont besoin de la valeur de la variable partagée, sont suspendus jusqu'au moment où le producteur l'instancie (voir le cas où le mode input n'est pas respecté, au point b ci-dessus).

Exemple : Appel : **r1(X),r2(X,true).**
Procédures : **mode r1(?).**
r1(A) :- ...

mode r2(^,?).
r2(X,Y) :- X = Y.

Le processus **r1** est consommateur de la variable partagée **X** car cette variable correspond à un argument de type input. Tant qu'elle n'a pas de valeur, l'unification du sous-objectif **r1(X)** de l'appel et de la tête de clause **r1(A)** est suspendue car elle ne respecte pas le mode input.

Le processus **r2** est producteur pour la variable **X**. Cette variable est de type output pour la relation **r2** et reçoit la valeur de **Y** (=true) dans le corps de la clause **r2(X,Y)**. Dès que la valeur est attribuée par **r2** à **X**, l'unification qui était suspendue pour **r1** peut reprendre.

L'attente sous forme de suspension pour une valeur d'une variable partagée est donc le moyen utilisé dans Parlog pour synchroniser les processus concurrents.

Il est important de signaler une caractéristique fondamentale des gardes de clauses Parlog. Les prédicats appartenant à de telles gardes peuvent seulement tester les valeurs de variables correspondant à des arguments de type input.

Ils ne peuvent générer aucune liaison pour des variables correspondant à des arguments de type input de l'appel. Si un prédicat d'une garde génère de telles liaisons, la garde n'est pas sûre. Une vérification à la compilation permet de détecter les gardes qui ne sont pas sûres. L'erreur étant signalée à la compilation, aucune garde n'exportera de liaisons illégales à l'appel durant l'exécution.

Il faut remarquer que générer une liaison pour une variable de la tête de clause qui est évaluée revient à générer une liaison pour la variable correspondante dans

l'appel. Ces liaisons seront donc reportées jusqu'au "commit", aucun des arguments de l'appel ne sera lié avant que la clause ne soit sélectionnée.

L'évaluation d'une garde peut seulement générer des liaisons pour :
1) des variables locales à la clause (n'apparaissant pas dans la tête de clause)
2) des variables qui apparaissent dans la tête de la clause, mais sont de type output.

d) Exécution du "commit"

Le "commit" n'est exécuté que pour une seule clause candidate appartenant à la procédure invoquée dans l'appel (cfr section I.2.8.).

C'est seulement au moment du "commit" que les liaisons générées par l'évaluation de la garde ou l'unification de la tête de clause pour les arguments de type output de l'appel sont effectuées. Aucune variable dans l'appel ne sera instanciée tant que le choix de la clause candidate n'est pas encore déterminé par le "commit" de cette clause.

Cette interdiction d'exporter une liaison à un argument de type output avant que le "commit" ne soit effectué est implémentée à la compilation. La première étape de la compilation consiste à *transformer le programme Parlog en une forme standard*, en se basant sur la déclaration de mode.

Un programme sous forme standard possède les caractéristiques suivantes : tous les arguments dans chaque tête de clause sont des variables, et les contraintes imposées par les modes sont traduites par des conditions d'unification explicite dans la garde ou le corps des clauses. Les contraintes liées au mode input sont traduites dans la garde, et celles liées au mode output, dans le corps. Les contraintes d'unification qui étaient implicites, dans les termes des arguments de tête de clause et dans la déclaration de mode, sont donc rendues explicites en ajoutant aux clauses des primitives d'unification.

Exemple de transformation sous forme standard :

◇ programme Parlog :

```
mode r1(?,^).
r1([X|Y],un) :- ...
r1([],deux) :- ...
```

◇ forme standard du programme Parlog :

r1(A,B) :- [X|Y] <= A | B := un ...
r1(A,B) :- [] <= A | B := deux ...

Dans la lecture logique du programme, '<=' et ':=' sont considérés comme des relations d'égalité. Au niveau de l'interprétation procédurale, '<=' suspend si son argument de droite est une variable qui devrait être instanciée afin de vérifier l'égalité. Quant à la primitive correspondant à ':=', elle effectue l'assignation (elle affecte à l'argument de gauche la valeur spécifiée par celui de droite) et elle ne réussit que si son argument de gauche est une variable non instanciée. Si tel n'est pas le cas, elle signale une erreur.

Le principal résultat à retenir de cette transformation de programme à la compilation est que des primitives correspondant au mode output sont ajoutées dans le corps de la clause et que l'unification correspondante peut être effectuée immédiatement après le "commit", quand la clause est devenue la clause candidate sélectionnée. Pour plus de détails sur les mécanismes d'unification de Parlog, le lecteur consultera [ClGr86], pp 11-14.

e) Exécution du corps

La conjonction de sous-objectifs formée par le corps est semblable à celle de la garde pour ce qui est de sa construction (opérateurs de conjonction ',' ou '&'), ainsi que de son évaluation (synchronisation des processus concurrents par mise à jour de variables partagées).

Comme expliqué au point d ci-dessus, lors de la compilation d'un programme Parlog, des primitives d'unification explicite sont ajoutées dans le corps des clauses. Ces primitives sont dès lors considérées comme des sous-objectifs supplémentaires dans la conjonction d'appels formée par le corps. Si l'exécution de ces primitives échoue, c'est aussi l'échec du corps de la clause et dès lors de la résolution de l'appel, puisqu'il n'y a pas de rétroparcours sur le choix de la clause (cfr remarque à ce sujet à la section I.2.9.).

Il y a néanmoins un *problème potentiel* avec cette présence des primitives d'unification dans le corps de la clause. Si plusieurs producteurs existent pour la même variable partagée dans le cadre d'une conjonction d'appels, il se peut que, entre le "commit" d'une clause **c1** et l'instanciation des variables de l'appel par celle-ci, une des variables soit liée par un autre processus producteur. L'exécution du corps de la clause **c1** ne se terminera pas sur une réussite ou un échec, mais bien sur une erreur car les conditions du mode output ne sont plus vérifiées (l'argument de l'appel n'est plus une variable car il a été instancié par un autre producteur).

Exemple : Appel : **r1(val1,X), r2(val2,X).**

Procédures : **mode r1(?,^).**
r1(Y,true) :- Y = val1 | ...

mode r2(?,^).
r2(B,false) :- B = val2 | ...

Transformation sous forme standard des deux procédures r1 et r2 :

r1(Y,Z) :- Y = val1 | Z := true ...

r2(B,A) :- B = val2 | A := false ...

Supposons que les deux clauses **r1(Y,Z)** et **r2(B,A)** fassent leur "commit" presque en même temps. Néanmoins, le processus correspondant à **r1** est plus rapide que celui correspondant à **r2** et est le premier à instancier la variable partagée **X** (via l'unification de **Z** et true). Quand le processus **r2** veut exécuter sa primitive d'unification, elle résulte en une erreur car la variable **X** a déjà reçu une valeur (**A** est instanciée à true), ce qui enfreint les règles imposées par le mode output.

I.3.4. Exemples de programmes Parlog

Les deux programmes décrits ci-dessous seront aussi repris comme exemples pour les langages Concurrent Prolog et GHC.

1) Fusion de deux flots de données

- *Spécification*

Il s'agit de fusionner deux flots de données **X** et **Y** en un troisième flot **Z**. Ce nouveau flot contiendra les éléments de **X** et de **Y**, et seulement ceux-ci, en préservant l'ordre relatif qu'ils avaient dans leur flot d'origine respectif.

La différence entre un flot et une liste est que le flot n'est que partiellement déterminé à tout moment de l'évaluation, alors que la liste est entièrement déterminée. Un flot est donc une liste construite de façon incrémentale.

• *Construction générale du programme*

Nous allons, pour construire le programme de fusion, nous baser sur les descriptions de la situation générale (où il faut encore itérer), et des conditions d'arrêt.

Les flots de données seront implémentés sous forme de listes.

Les éléments de chacune des deux listes X et Y seront considérés un à un. Trois arguments seront nécessaires pour la procédure de fusion, ils correspondront chacun à une des trois listes (X, Y et Z). Les deux arguments X et Y doivent être instanciés avant de commencer le traitement. Si tel n'est pas le cas, une suspension sera effectuée jusqu'au moment où une valeur sera assignée aux listes X et Y. La façon dont cette suspension est prescrite étant propre à chacun des trois langages, elle ne sera pas abordée dans le cadre de cette construction générale.

La *situation générale* correspond à la situation où il y a encore des éléments à traiter dans les deux listes. Deux cas peuvent alors être envisagés.

1er cas : on assigne au premier élément de la liste résultat Z la valeur du premier élément de la liste X, et on appelle récursivement la procédure de fusion avec comme arguments la queue de la liste X (le premier élément a été traité), la liste Y inchangée, et la queue de la liste Z (le premier élément a reçu une valeur). On se rapproche ainsi de la situation finale où tous les éléments de X et de Y sont traités (se retrouvent dans la liste Z), car le nombre d'éléments restant à traiter a diminué d'une unité.

2ème cas : on assigne au premier élément de la liste Z la valeur du premier élément de la liste Y, et on appelle récursivement la procédure de fusion avec comme arguments la liste X inchangée, la queue de la liste Y (il ne faut plus considérer le premier élément), et la queue de la liste Z. Tout comme pour le premier cas, on se rapproche ainsi de la situation finale.

Deux clauses seront nécessaires pour décrire ces deux cas d'itération. Le non-déterminisme lié à l'existence du parallélisme-OU ne permettant pas de savoir quel cas sera pris en compte, il faudra veiller à ce que la liste Z ne soit pas modifiée avant que le choix du cas traité ne soit définitif.

Les *conditions d'arrêt* sont au nombre de deux.

1) La fin de la liste X est atteinte.

Cette situation est caractérisée par la valeur de l'argument X : celui-ci est instancié à la liste vide car tous les éléments ont été traités. Il suffit dans ce cas de donner aux éléments restants de Z les valeurs des éléments restants de la liste Y. L'argument Z se verra donc assigner la valeur de l'argument Y. Aucune itération supplémentaire n'est plus nécessaire car cette situation correspond à la situation finale : la liste Z comprend tous les éléments de X et de Y, et eux seuls.

2) La fin de la liste Y est atteinte.

Cette seconde condition d'arrêt est exactement la symétrique de la première.

Deux clauses seront nécessaires pour décrire ces deux conditions d'arrêt. Dans le cas où les deux conditions sont vérifiées en même temps, le non-déterminisme ne permet pas de dire quelle clause sera utilisée. L'assignation d'une valeur à la liste Z ne pourra donc être effectuée qu'après le choix de la clause. De cette façon, quel que soit le choix, l'exécution se terminera correctement.

Les quatres clauses de la procédure de fusion peuvent être évaluées en parallèle-OU sans risque de non-terminaison car dans le cas où les deux listes X et Y sont vides (condition d'arrêt), seules les clauses correspondant aux conditions d'arrêt peuvent être candidates.

Le programme ainsi construit vérifie sa spécification car tous les éléments de X et Y, et eux seuls, se trouvent repris dans la liste Z. De plus, comme les éléments sont considérés dans l'ordre où ils apparaissent dans les deux listes d'entrée, leur ordre relatif est préservé.

L'exécution du programme sera suspendue dès le début dans le cas où les arguments correspondant aux listes X et Y ne sont pas instanciés avant le début du traitement. Aucun cas de suspension n'est prévu durant l'exécution. Des problèmes de non-terminaison de l'exécution ne devraient donc pas se poser.

Les résultats obtenus par la construction générale de ce programme de fusion doivent maintenant être particularisés, de façon à obtenir les programmes en Parlog, Concurrent Prolog ou GHC. Cette construction générale ne sera pas reprise dans le cadre des exemples en Concurrent Prolog et en GHC.

• *Construction propre à Parlog*

Les deux arguments de la procédure Parlog correspondant aux deux listes X et Y sont de type input. Ainsi, dans le cas où une des deux listes représentant les flots d'entrée n'est pas instanciée, l'évaluation sera suspendue jusqu'au moment où une valeur sera assignée au flot concerné.

Aussi bien dans le cas des itérations que dans celui des conditions d'arrêt, il est important que l'élément de Z ne soit pas instancié avant que le choix de la clause utilisée ne soit effectué. C'est pourquoi l'argument correspondant à Z est de type output. Son instanciation sera ainsi reportée pour n'être exécutée qu'au début du corps de la clause qui aura été sélectionnée (cfr point d de la section I.3.3.).

• *Code du programme*

```
    mode merge(?,?,^).
(1) merge([Xs|X],Y,[Xs|Z]) :- merge(X,Y,Z).
(2) merge(X,[Ys|Y],[Ys|Z]) :- merge(X,Y,Z).
(3) merge(X,[],X).
(4) merge([],Y,Y).
```

2) Système de réservation aérienne

• *Spécification*

Ce problème ("jouet") a été résolu pour Concurrent Prolog par Shapiro dans [Sh83], sur base de la description qui en avait été faite par Bryant et Dennis (cfr [BrDe82]).

Le système de réservation aérienne dispose d'une base de données contenant l'information sur les vols d'une ligne aérienne. Initialement, chaque vol a 100 places disponibles. Le système peut accepter deux types de commandes :

1) de la part d'un agent :
réserver n places sur un vol f.

- si au moins n places sont disponibles sur le vol f, les places seront réservées et le système répondra avec le message 'true'.
- s'il n'y a pas assez de places libres, le système répondra avec le message 'false' sans rien modifier à la base de données.

Dans ce cas, la commande suivante ne sera traitée que si la mise-à-jour de la base de données a été effectuée sans problème. Dans le cas où une erreur technique de manipulation de base de données survenait durant la mise-à-jour et rendait celle-ci impossible, le traitement devrait être arrêté car il n'y aurait plus aucune garantie quant à la validité du contenu de la base de données.

2) de la part d'un utilisateur du système :
trouver combien de places sont disponibles sur le vol f.

Le système répondra avec le nombre de places libres sur le vol f au moment où la commande est traitée. Ce nombre de places ne correspondra pas nécessairement à la situation réelle (il peut y avoir une mise-à-jour entre le moment où la commande est prise en compte et le moment où la réponse est donnée).

Les commandes sont adressées au système sous la forme d'un flot de commandes.

• *Construction générale du programme*

A partir de la description de la situation générale, certains sous-problèmes vont être identifiés, avant d'être raffinés. Les résultats ainsi obtenus, couplés à la description de la condition d'arrêt, permettront d'obtenir les lignes générales du programme de réservation aérienne.

Le flot de commandes sera implémenté sous la forme d'une liste. Chacun des éléments de cette liste correspondra à une commande. La procédure principale (appellée database) aura deux arguments : la liste correspondant au flot de commandes (S) et la base de données (DB). La liste S doit absolument être instanciée avant d'effectuer tout traitement. Si tel n'est pas le cas, une suspension devra être effectuée en attendant qu'elle ait reçu une valeur. Nous verrons dans le cadre de chacun des trois langages comment cette suspension peut être prescrite.

La *situation générale* est la situation dans laquelle il reste encore des commandes non traitées dans la liste, toutes les commandes déjà traitées ayant laissé la base de données dans un état cohérent. Cet état est tel que pour tout vol, le nombre de places libres est ≥ 0. Deux cas peuvent alors se présenter : l'élément suivant dans la liste de commandes est une commande de consultation, ou est une commande de réservation.

1er cas : le premier élément de la liste est une commande de consultation. Une telle commande est définie par l'atome 'info' et deux variables : la première (V) instanciée à la valeur d'un numéro de vol de la base de données, et la seconde (P) à instancier à la valeur du nombre de places libres sur ce vol.
Il faut consulter la base de données DB pour assigner à P la valeur de l'élément correspondant au vol de numéro V dans la base de données. Le prédicat correspondant à cette consultation est le prédicat 'value'. Il a trois arguments : DB, V, P. Son rôle est d'assigner à P la valeur de l'élément correspondant au vol V dans la base de données DB. Il faut également appeller récursivement la procédure database avec pour arguments la queue de la liste de commandes (sans le premier élément qui vient d'être traité) et la base de données inchangée.
Les deux prédicats correspondant respectivement à la consultation (value) et à l'appel récursif (database) peuvent être résolus en parallèle-ET sans synchronisation spéciale. Cela est dû au fait que les variables qu'ils partagent (le flot de commandes et la base de données) ne sont qu'accédées en lecture, aucune modification n'est réalisée. Dans le cas où l'appel récursif est résolu avant que la consultation ne le soit, il est possible que la commande suivante à traiter soit une réservation. Si cette réservation est effectuée, l'itération suivante considèrera la base de données mise à jour. La base de données initiale n'est cependant pas modifiée. Le résultat de la consultation ne sera donc pas influencé par la résolution en parallèle-ET de l'appel récursif. L'état de la base de données considéré par la

consultation ne comprend donc peut-être pas les dernières modifications enregistrées.

On s'est ainsi rapproché de la situation finale où toutes les commandes ont été correctement traitées car le nombre de commandes restant à traiter a diminué d'une unité.

Ce cas sera décrit par une des clauses de la procédure principale database.

2ème cas : le premier élément de la liste de commandes est une commande de réservation. Une telle commande est définie par l'atome 'reserve' et trois variables : la première (V) est instanciée à la valeur du numéro de vol pour lequel il faut réserver des places, la deuxième (P) est instanciée à la valeur du nombre de places à réserver et la troisième (R) sera instanciée à 'false' ou à 'true' en fonction du résultat de la réservation.

Il faut effectuer la réservation en fonction des valeurs de V et de P, et du contenu de la base de données DB. Si le nombre de places libres était suffisant pour effectuer la réservation, la valeur 'true' est assignée à la variable R et dans le cas contraire, c'est la valeur 'false' qui lui est assignée. Ce sous-problème de réservation (appellé reserve) sera raffiné dans la suite de la construction du programme.

Si l'exécution de la réservation (la résolution de l'appel reserve) a été menée à son terme, on appelle récursivement la procédure database avec pour arguments la queue de la liste de commandes et la base de données dans l'état où elle se trouve après la réservation. On s'est ainsi rapproché de la situation finale, car le nombre de commandes restant à traiter a diminué de un.

Sinon, on se trouve dans le cas d'une erreur technique de manipulation de base de données, l'exécution doit dès lors se terminer. Il faut donc s'assurer qu'on n'itèrera pas avec la base de données si la réservation ne se termine pas bien. C'est pourquoi l'appel au prédicat reserve sera effectué dans la garde de la clause correspondant à ce cas, et l'itération avec appel récursif à database, dans le corps. Comme le corps de la clause n'est pas exécuté si l'évaluation de la garde échoue, et comme il n'y a qu'une clause traitant le cas d'une commande de réservation, l'échec de sa garde entraîne l'échec de l'appel initial (aucune autre clause candidate n'étant disponible pour être sélectionnée). L'exécution se termine donc sur une erreur.

Les deux clauses correspondant aux cas identifiés ci-dessus décrivent chacune un type de commande de façon unique. En présence d'une certaine commande, le non-déterminisme est donc éliminé, car la clause traitant la commande est unique. Le parallélisme-OU ne crée donc pas de problème.

Le *raffinement du sous-problème reserve* va maintenant être effectué. La procédure reserve aura cinq arguments : le numéro de vol (V), le nombre de places à réserver (P), la base de données initiale (DB), la variable de réponse devant être

instanciée en fonction du résultat de la réservation (R), et la base de données après réservation (DB1).

L'effet de la procédure reserve peut être décrit de la façon suivante . En fonction du nombre de places libres pour le vol V dans la base de données initiale DB, essayer de réserver P places sur ce vol. Si la réservation peut être effectuée, assigner la valeur 'true' à la variable R et modifier la base de données pour obtenir le contenu mis-à-jour à assigner à DB1. Si tel n'est pas le cas, assigner la valeur 'false' à R, et le contenu inchangé de DB à DB1.

Trois sous-problèmes peuvent être identifiés à partir de cette description :

1) la consultation de la base de données DB pour obtenir dans la variable D la valeur du nombre de places libres pour le vol V. Le prédicat remplissant cette fonction est le prédicat value déjà utilisé dans le premier cas de la situation générale.

2) le calcul du nombre de places restant libres après la réservation, soit L (= D - P),

3) l'envoi de la réponse de la réservation : (le prédicat correspondant à cette primitive est appellé response, et sera raffiné ci-dessous)

Si le nombre de places restant libres (L) est ≥ 0, la modification est effectuée : on instancie la variable R à 'true' et on assigne à la variable DB1 le contenu mis-à-jour de la base de données DB (où la valeur de l'élément correspondant au vol V est remplacée par la valeur de L).

Sinon, on instancie la variable R à 'false' et la variable DB1 au contenu inchangé de la base de données DB. •

Les trois prédicats du corps de la clause de la procédure reserve (value, calcul et response) pouvant être résolus en parallèle-ET, il est nécessaire que les processus correspondant à leur résolution puissent se synchroniser. Cette synchronisation sera effectuée par mise-à-jour de variables partagées (ici, D et L). La résolution du prédicat de calcul doit être suspendue en attendant que celle du prédicat value ait instancié la variable qui leur est commune (D). De la même façon, la résolution du prédicat response est suspendue tant que la valeur de la variable que ce prédicat partage avec le prédicat de calcul (L) n'est pas instanciée par ce dernier. Le processus de calcul est consommateur pour la variable D, tandis que le processus value est producteur pour cette même variable. En ce qui concerne la variable partagée L, le processus de calcul en est producteur et le processus response, consommateur. Nous verrons comment les trois langages logiques parallèles considérés implémentent cette synchronisation.

La synchronisation ainsi décrite ne devrait pas poser de problème de non-terminaison car :

1) à tout consommateur correspond un producteur;

2) le producteur response ne sera jamais bloqué, il pourra donc toujours fournir une valeur au consommateur (calcul), celui-ci ne sera donc pas bloqué indéfiniment et pourra tôt ou tard remplir son rôle de producteur;

3) aucune coalition n'est possible entre consommateurs puisqu'il n'y a qu'un seul consommateur par producteur.

Le *raffinement du sous-problème response* sera basé sur la description de la fonction à réaliser par la résolution de response proposée dans le cadre du raffinement du sous-problème reserve.

La procédure response devra comporter 5 arguments : la base de données (DB), le nombre de places restant libres après la réservation (L), le numéro du vol (V), la variable (R) à instancier en fonction du résultat de la réservation, et la variable DB1 à instancier au contenu de la base de données à utiliser pour la suite du traitement.

Les variables R et DB1 seront instanciées en fonction des deux cas suivants :

1) il reste des places en suffisance ($L \geq 0$)

Dans ce cas, il faut effectuer la modification de la base de données en fonction des valeurs des variables DB, L et V, de façon à pouvoir assigner à la variable DB1 la valeur de la base de données mise-à-jour. La valeur 'true' est assignée à la variable R.

2) il n'y a plus assez de places ($L < 0$)

La base de données n'étant pas mise-à-jour, la variable DB1 se voit assigner la valeur du contenu inchangé de DB. La valeur 'false' est assignée à la variable R.

Deux clauses seront nécessaires pour décrire les deux cas identifiés ci-dessus.

Pour un certain nombre de places restant libres, seulement une clause peut être candidate. De façon à n'exécuter le corps de la clause que si celle-ci correspond à la description de la situation de réservation, il est indispensable d'effectuer le test de comparaison de la valeur de L avec 0 durant l'évaluation des deux clauses. Ce test sera donc effectué dans la garde de chaque clause. Le non-déterminisme n'intervient pas car le choix de la clause est ainsi imposé. Il faudra veiller à ce que rien ne soit exporté comme liaison à l'appel tant que la clause candidate n'a pas franchi l'opérateur de "commit".

La *condition d'arrêt* correspond à la situation où toutes les commandes ont été traitées, l'argument correspondant au flot de commandes restant à traiter est instancié à la liste vide. Une clause de la procédure principale décrira ce cas : si la condition d'arrêt est vérifiée, il n'y a plus rien à faire car le traitement est terminé.

Les trois clauses de la procédure principale peuvent être évaluées en parallèle-OU sans pour autant entraîner l'existence d'un non-déterminisme quant au choix d'une clause candidate. En effet, à chaque situation (1er cas de la situation générale, 2ème cas de la situation générale, condition d'arrêt) ne correspond qu'une seule clause. Le choix de celle-ci est donc imposé.

La procédure principale ainsi décrite se terminera car aucune suspension indéfinie ne devrait se produire durant l'exécution (mise à part l'éventuelle suspension au départ si le flot de commandes n'est pas instancié). L'exécution de cette procédure

verifie la spécification du problème : les commandes sont correctement traitées et la base de données reste dans un état cohérent.

Il faut encore particulariser les résultats obtenus par cette construction générale aux trois langages considérés dans le cadre de ce travail. Ces résultats généraux ne seront par repris dans les constructions propres à Concurrent Prolog et GHC.

• *Construction propre à Parlog*

L'argument du flot de commandes de la procédure principale devant absolument être instancié avant d'effectuer tout traitement, il sera décrit comme étant de type input.

En ce qui concerne la synchronisation des trois prédicats de la procédure reserve, elle est effectuée comme suit. L'argument (D) partagé par les deux prédicats value et calcul sera de type output pour le prédicat value, et de type input pour celui de calcul. La garde de la clause du prédicat de calcul fait appel au prédicat prédéfini data(X). Ce prédicat ne réussit que si son argument (ici, D) n'est pas une variable non instanciée. Dans le cas contraire, il est suspendu. Le processus correspondant à la résolution du prédicat de calcul sera donc suspendu tant que l'argument D restera une variable non instanciée. Dès que le processus de consultation aura instancié D, l'évaluation de la clause pour le calcul pourra reprendre.
L'utilisation du prédicat data(X) pour spécifier une suspension est nécessaire, car l'unification d'un argument variable de type input dans l'appel avec une variable dans la tête de clause réussit. Afin de suspendre cette unification tant que l'argument de l'appel n'est pas instancié, il faut ajouter une condition. En effet, ce cas de suspension n'est pas repris dans la sémantique associée au mode input d'un argument. Le prédicat data(X) est une des primitives d'unification nécessaires pour Parlog.
Les deux processus correspondant à la résolution du prédicat calcul et du prédicat response sont synchronisés de la même façon grâce à la variable partagée L.

Pour la procédure response, il est nécessaire de déclarer de type output les deux arguments correspondant aux variables qui doivent être instanciées durant l'exécution (R et DB1). De cette façon, la transformation du programme à la compilation garantit que les instanciations ne seront effectuées que lorsque la clause candidate aura franchi l'opérateur de "commit".

• *Code du programme*

```
    mode database(?,?).
(1) database([info(Flight,Seats)|S],DB) :-
            value(DB,Flight,Seats),
            database(S,DB).
(2) database([reserve(Flight,Seats,Response)|S],DB) :-
            reserve(Flight,Seats,DB,Response,DB1) |
            database(S,DB1).
(3) database([],_).

    mode reserve(?,?,?,^,^).
(1) reserve(Flight,Seats,DB,Response,DB1) :-
          value(DB,Flight,Freeseats),
          plus(Seats,Leftseats,Freeseats),
          respond(DB,Leftseats,Flight,Response,DB1).

    mode respond(?,?,?,^,^).
(1) respond(DB,Seats,Flight,true,DB1) :- le(0,Seats) |
        modify(DB,Flight,Seats,DB1).
(2) respond(DB,Seats,_,false,DB) :- lt(Seats,0) | true.

   mode plus (?,^,?).
plus(X,Y,Z) :- data(X), data(Z) | Z is X + Y.

    mode le (?,?).
le(X,Y) :- data(X), data(Y) | X =< Y.

    mode lt(?,?).
lt(X,Y) :- data(X), data(Y) | X < Y.
```

I.4. Le langage CONCURRENT PROLOG

L'approche de ce langage est essentiellement basée sur [Sh83].

I.4.1. Syntaxe

• Les *variables* sont des noms commençant par une lettre majuscule, les atomes et prédicats commencent par des lettres minuscules.

• Un programme en Concurrent Prolog est un ensemble fini de *clauses gardées* de la forme suivante :

H :- G1, ..., Gm | B1, ..., Bn.

L'opérateur ' | ' est omis si m = 0 (cfr section I.2.6.).

• Dans une telle clause, les variables peuvent être accompagnées d'une information de contrôle spéciale, une *annotation "read-only"*, qui contrôle l'unification. Une annotation "read-only" est représentée par un "?" et est attachée à un terme **X** par la notation **X?**.

L'unification d'un terme annoté "read-only" **X?** avec un terme **Y** est définie comme suit.

(1) Si **Y** est une variable non instanciée, alors l'unification de **X?** et **Y** réussit.
  (1.1) Si X est une variable non instanciée, le résultat est une variable "read-only" (la propriété de "read-only" est transmise à la variable Y).
  (1.2) Si X est instanciée, le résultat de l'unification est l'instanciation de Y à la valeur de X.

(2) Sinon,
  (2.1) Si X est une variable instanciée, l'unification réussit si et seulement si l'unification de X et de Y réussit.
  (2.2) Si X est une variable non instanciée, l'unification est suspendue jusqu'à ce que X devienne instanciée. Il faut alors appliquer (2.1).

Cette définition de l'unification de Concurrent Prolog a comme particularité que des variables apparaissant dans un terme "read-only" ne sont pas nécessairement "read-only". En effet, dans le cas d'un terme composé, la portée de l'annotation "read-only" est le foncteur du terme, et pas ses arguments.

Exemple :

- soit le terme "read-only" X? dont la valeur est f(A,B).
- soit Y = f(V,W).
- l'unification de X? et Y réussit car ils sont tous deux des termes non variables et sont unifiables récursivement.
- le résultat de l'unification est que les variables A et V "partagent" (cfr section I.2.3.), tout comme B et W. Il est important de signaler qu'elles ne sont pas des variables "read-only" puisque A et B n'avaient pas hérité de cette propriété (seul le foncteur 'f' était concerné).

Tout comme proposé dans [Dv84], les différents cas de l'unification d'une variable annotée "read-only" X? avec une variable Y peuvent être résumés sous forme de tableau.

valeur de : (à l'exécution) **X** / **Y**	**non instanciée**	**instanciée à T1**
non instanciée	réussite Y = X?	réussite Y = T1
instanciée à T2	suspension	réussite ou échec selon que T1 et T2 sont unifiables ou non

Dans le cas particulier de l'unification de deux variables annotées "read-only", Shapiro prévoit une suspension (cfr [Ue85b]).

Le rôle de ces annotations "read-only" sera abordé dans la section suivante.

I.4.2. Sémantique procédurale

a) Ordre d'essai des clauses

Les clauses d'une même procédure forment un ensemble dans lequel l'ordre est non significatif.

La place que chaque clause occupe dans le texte de la procédure n'a aucune influence sur l'ordre dans lequel les clauses sont évaluées pour résoudre un appel donné : l'évaluation est réalisée en parallèle-OU.

b) Unification de tête de clause

Certaines contraintes sont imposées pour l'unification d'un appel et d'une tête de clause par la présence d'annotations "read-only" sur certaines variables.

L'unification d'un sous-objectif d'un appel où apparaissent des variables annotées "read-only" et d'une tête de clause sera suspendue si on essaie d'instancier avec un terme non variable une variable annotée "read-only". La clause pour laquelle l'unification était effectuée passe donc dans l'état suspendue. Si la variable est partagée par un autre sous-objectif de l'appel (pour lequel elle n'est pas annotée "read-only"), celui-ci peut l'instancier à un terme non variable. L'unification qui avait été suspendue pourra alors reprendre.

Puisqu'une unification qui doit être suspendue à un moment donné peut réussir plus tard, l'unification peut être interprétée comme une activité continue, qui ne se termine que sur un succès ou un échec définitif (un échec est définitif quand il n'est pas dû à la violation de contraintes "read-only").

Exemples :

1. appel : **r(A?,B).**
   clause : **r(X,valB)** :- ...

L'unification de la tête de clause avec l'appel réussira en donnant comme résultat : **X** est devenue une variable annotée "read-only" et "partage" avec **A**; **B** est instanciée à la valeur du second argument de la tête de clause (= valB). Il n'y a aucun problème pour l'instanciation de **B** car ce n'est pas une variable annotée "read-only".

2. appel : **r1(A?,B), r2(A).**
   clauses : **r1(valA,Z)** :- ... (1)
   r2(valA) :- ... (2)

L'unification de la tête de clause (1) avec le sous-objectif **r1** de l'appel sera suspendue car on essaie d'instancier la variable annotée **A?** avec une constante (valA). Par contre, l'unification de la tête de clause (2) avec le second sous-objectif de l'appel réussira car dans cet objectif, la variable partagée **A** n'est pas annotée "read-only". Elle peut donc être instanciée à la valeur 'valA'. Dès que **A** est instanciée à un terme non variable (ici, l'atome valA), on peut reprendre l'évaluation de la clause (1). Celle-ci n'est donc plus suspendue et la suite de l'évaluation permettra de déterminer si elle est candidate ou non.

c) Evaluation de la garde

Les sous-objectifs de la garde sont évalués en parallèle-ET. La communication entre les processus correspondant aux évaluations de ces sous-objectifs est réalisée par la mise à jour de variables communes.

La synchronisation entre producteur et consommateurs d'une variable partagée est effectuée de la façon suivante. Les processus consommateurs (processus qui veulent utiliser la valeur de la variable) correspondent à des sous-objectifs pour lesquels la variable est annotée "read-only". Ces processus seront donc suspendus jusqu'au moment où le producteur instanciera la variable à un terme non variable (voir l'exemple 2 du point b ci-dessus).

Les processus concurrents sont donc synchronisés grâce à l'attente (par suspension) d'une valeur pour une variable qui leur est commune.

Des problèmes peuvent apparaître si plusieurs producteurs existent pour une même variable. Il faut alors faire une vérification de consistance de la valeur de la variable partagée. La proposition est faite dans [TaFu83] de remplacer ces vérifications par la restriction que le processus producteur d'une valeur pour une variable partagée soit unique. L'identité de ce processus ne doit pas nécessairement être connue avant l'exécution, le processus pouvant être déterminé dynamiquement de façon non déterministe. Rien n'est suggéré quant à un moyen d'implémenter cette restriction. Il est à remarquer que cette limitation élimine tout problème d'instanciation d'une même variable par plusieurs processus producteurs. Shapiro n'impose pas cette restriction dans [Sh83], le problème reste donc présent (voir le point d ci-dessous).

L'évaluation de la garde d'une clause ne peut débuter qu'après que l'unification de la tête de la clause avec l'appel ait réussi. Tant que cette unification est suspendue, la garde ne peut pas être évaluée.

Aucune restriction n'est faite en ce qui concerne le type des sous-objectifs de la garde. Les sous-objectifs peuvent spécifier des exportations de liaisons pour certaines variables de l'appel.

Dans le cas où la garde d'une clause est vide, la clause devient candidate dès que l'unification de sa tête avec l'appel a réussi.

d) Exécution du "commit"

Comme expliqué à la section I.2.8., le "commit" n'est exécuté que pour une seule clause candidate dans le cadre de la résolution d'un appel. Il faut donc garantir dans l'implémentation du langage que, si plusieurs clauses sont prêtes à franchir le "commit", alors seulement une clause recevra la permission de le faire.

C'est seulement après avoir reçu cette permission que les liaisons aux variables de l'appel générées durant l'évaluation de la garde de la clause seront effectuées. Rien ne sera exporté à l'appel tant que le choix de la clause candidate n'est pas fixé. Comme on ne peut pas garantir avant l'exécution qu'aucune garde ne liera des variables dans l'appel, les arguments de l'appel doivent être copiés pour chaque clause qui est essayée pour résoudre l'appel. Ce mécanisme de copies est implémenté grâce à la *gestion d'environnements multiples*. Ces environnements sont locaux aux gardes de chacune des clauses de la procédure concernée par l'appel. Pour chaque garde des clauses essayées, des copies locales des variables de l'appel sont créées. Les modifications par instanciations seront donc effectuées sur ces copies locales. Il faut, au moment du "commit", unifier ces copies locales de variables avec les variables globales correspondantes. Cette unification pourrait échouer si d'autres processus concurrents ont instancié différemment ces variables, et ce, avant l'exportation de liaisons du processus considéré ici (cfr problème évoqué au point c ci-dessus).

Exemple : appel : **r1(X,Y), r2(Y,Z).**
clauses : **r1(valX,valY1)** :- ...
r2(Y,true) :- **Y = valY2** | ...

Si le processus correspondant au sous-objectif **r2** de l'appel est exécuté avant que la clause utilisée pour la résolution de **r1** ne fasse son "commit", **Y** a déjà reçu la valeur 'valY2'. Au "commit" de **r1**, on va exporter la liaison Y=valY1 mais cette unification va échouer car **Y** est déjà instancié à 'valY2'.

Il faut remarquer qu'avec la restriction signalée par [TaFu83], ce problème ne se pose pas.

Shapiro ne dit pas clairement quelle est la conséquence d'un échec de cette unification. Il dit seulement que si elle réussit, alors le "commit" se termine avec succès. Peut-être cette affirmation signifie-t-elle que si l'unification échoue, alors le "commit" se termine sur un échec. Mais le "commit" étant un prédicat qui réussit toujours, cette supposition est mauvaise. Une autre supposition quant à l'échec de l'unification serait de disposer d'une implémentation telle qu'après le "commit", on effectue l'unification des copies locales aux variables de l'appel. Si cette unification échoue, c'est l'échec de la résolution du corps de la clause et dès lors aussi celui de l'appel. Cette supposition semble être raisonnable car elle ne remet en question ni le rôle du "commit", ni la détection d'un échec éventuel de l'appel.

e) Exécution du corps

Comme vu au point précédent, la première action exécutée après le "commit" d'une clause est l'unification des variables de l'environnement local et des variables de l'appel. C'est seulement après la réussite de cette unification que le corps de la clause peut être exécuté. L'échec de cette unification équivaut à l'échec de la résolution du corps lui-même. L'unification des copies locales et des variables publiques ne doit pas nécessairement être une opération atomique. En effet, si différents processus essaient d'unifier les mêmes variables simultanément, alors leur succès ou leur échec est indépendant de l'ordre dans lequel les unifications sont faites.

Exemple : appel : **r1(X,Y), r2(Y,Z).**
clauses : **r1(A,B) :- B = valY1 | ...**
r2(P,Q) :- P = valY2 | ...

Il est clair que l'appel est voué à l'échec puisque les deux sous-objectifs veulent instancier la variable partagée **Y** à des valeurs qui sont incompatibles. Si les deux clauses utilisées pour la résolution font leur commit presque simultanément, on pourrait imaginer la séquence suivante pour ce qui est de l'exportation des valeurs des variables locales à la variable de l'appel **Y** :

1) le processus correspondant à **r1** exporte 'valY1' vers **Y**, l'unification réussit car **Y** n'était pas instanciée.

2) le processus correspondant à **r2** exporte 'valY2' vers **Y**. L'unification échoue car **Y** était déjà instanciée à 'valY1' ≠ 'valY2'. L'exécution du corps de la clause **r2** se termine sur un échec, le sous-objectif **r2(Y,Z)** échoue et la conjonction de l'appel se termine sur un échec.

Le processus qui découvre l'échec de la conjonction de l'appel est celui pour lequel l'unification de ses variables locales avec les variables de l'appel échoue. Son identité n'a pas d'importance car de toutes façons, le résultat sera le même : l'échec de l'appel. Les unifications peuvent donc être effectuées dans un ordre non déterminé.

Les différents sous-objectifs du corps de la clause qui a passé le "commit" sont résolus en parallèle-ET, tout comme ceux de la garde (cfr point c ci-dessus). La synchronisation entre les processus est réalisée par instanciation de variables partagées. Les processus consommateurs sont suspendus tant que le producteur n'a pas assigné une valeur à la variable commune considérée. Ce sont les annotations "read-only" qui permettent de gérer cette suspension (cfr point b ci-dessus).

1) Fusion de deux flots de données

La spécification et la construction générale sont les mêmes que celles qui ont été données pour ce programme en Parlog (cfr section I.3.4.).

• *Construction propre à Concurrent Prolog*

Les arguments correspondant aux listes X et Y sont annotés "read-only" dans l'appel, de façon à suspendre l'évaluation quand les deux listes ne sont pas instanciées. Dans les deux clauses décrivant la situation générale, il est nécessaire pour l'objectif d'itération d'annoter "read-only" l'argument correspondant à la queue de la liste dont on vient de traiter le premier élément. Cela s'impose pour suspendre l'évaluation dans le cas où les deux listes ne sont pas instanciées. Il n'est pas nécessaire d'annoter l'argument correspondant à la seconde liste d'entrée car celui-ci a hérité de l'annotation de l'argument de l'appel. Il est donc devenu lui aussi "read-only" (voir règle de transmission de la propriété de "read-only" à la section I.4.1.).

Le mécanisme d'environnements multiples (cfr point d de la section I.4.2.) permet aux instanciations de n'être effectuées qu'après le choix de la clause utilisée pour la résolution. Le non-déterminisme existant au niveau du choix d'une clause candidate ne pose donc pas de problème.

• *Code du programme*

```
(1) merge([Xs|X],Y,[Xs|Z]) :- merge(X?,Y,Z).
(2) merge(X,[Ys|Y],[Ys|Z]) :- merge(X,Y?,Z).
(3) merge(X,[],X).
(4) merge([],Y,Y).
```

2) Système de réservation aérienne

La spécification et la construction générale de ce programme ont déjà été présentées pour Parlog à la section I.3.4. Seules seront abordées ici les caractéristiques propres à Concurrent Prolog.

• *Construction propre à Concurrent Prolog*

L'argument du flot de commandes de la procédure principale devant être instancié pour effectuer le traitement, il sera annoté "read-only" dans l'appel. Lors de l'appel récursif avec la suite de la liste de commandes, il faudra à nouveau annoter cet argument, de façon à prescrire une suspension si le flot de commandes n'est pas instancié.

En ce qui concerne la synchronisation des processus de résolution des trois prédicats de la procédure reserve, des suspensions sont nécessaires quand un consommateur doit attendre une valeur pour la variable qu'il partage avec le producteur. Les suspensions sont prescrites à partir des gardes des clauses des procédures consommateurs des variables : celles-ci comportent, de façon directe ou indirecte, un appel au prédicat prédéfini wait(X). Ce prédicat prescrit une attente jusqu'au moment où son argument est instancié. Dès que l'argument a reçu une valeur, le prédicat se termine. Le prédicat wait(X) constitue une extension de la primitive d'unification de Concurrent Prolog. Il est semblable au prédicat data(X) pour Parlog.

Aucun problème ne se pose pour la procédure response car la présence d'environnements multiples garantit que rien n'est exporté à l'appel tant que la clause n'est pas sélectionnée par passage du "commit".

• *Code du programme*

```
(1) database([info(Flight,Seats)|S],DB) :-
          value(DB,Flight,Seats),
          database(S?,DB).
(2) database([reserve(Flight,Seats,Response)|S],DB) :-
          reserve(Flight,Seats,DB,Response,DB1) |
          database(S?,DB1).
(3) database([],_).

(1) reserve(Flight,Seats,DB,Response,DB1) :-
          value(DB,Flight,Freeseats),
          plus(Seats,Leftseats,Freeseats),
          respond(DB,Leftseats,Flight,Response,DB1).

(1) respond(DB,Seats,Flight,true,DB1) :-
          le(0,Seats) | modify(DB,Flight,Seats,DB1).
(2) respond(DB,Seats,_,false,DB) :-
          lt(Seats,0) | true.

le(X,Y) :- wait(X), wait(Y) | X ≤ Y.
lt(X,Y) :- wait(X), wait(Y) | X < Y.
plus(X,Y,Z) :- wait(X), wait(Z) | Z is X + Y.
```

I.5. Le langage GUARDED HORN CLAUSES

La description de ce langage est basée sur l'article [Ue85a].

I.5.1. Syntaxe

• Les *variables* sont des noms commençant par une lettre majuscule, les atomes et prédicats commencent par des lettres minuscules.

• Une procédure GHC est un ensemble fini de *clauses gardées* de la forme suivante :

H :- G1, ..., Gm | B1, ..., Bn. avec m,n ≥ 0.

L'opérateur ' | ' est appellé opérateur de "trust".
La partie à gauche de cet opérateur (H + {Gi}) est appellée partie passive de la clause.
La partie à droite ({Bi}) est appellée partie active ou corps de la clause.
Le prédicat vide 'true' est utilisé pour noter une garde ou un corps vide (m ou n = 0) (cfr section I.2.7.).

• La syntaxe de GHC ne comporte *aucune construction spéciale concernant le contrôle du parallélisme*.

I.5.2. Sémantique procédurale

a) Ordre d'essai des clauses

Les différentes clauses d'une procédure correspondant à l'appel à résoudre sont essayées de façon parallèle-OU. L'ordre dans lequel elles sont évaluées est non significatif.

b) Unification de tête de clause

Toute unification d'un appel et d'une tête de clause qui aurait pour effet de lier une variable de l'appel en lui exportant une valeur sera suspendue. L'unification pourra reprendre quand un autre processus concurrent aura instancié la variable concernée. Le résultat de cette unification sera une réussite si la valeur attribuée par le second processus est unifiable avec la valeur spécifiée dans la tête de clause. Dans le cas contraire, le résultat sera un échec.

Exemple : appel : **r1(X), r2(X).**

clauses : **r1(val1)** :- ... (1)

r1(val2) :- ... (2)

r2(Y) :- true | Y= val1. (3)

L'unification de la tête de la clause (1) avec le sous-objectif **r1(X)** de l'appel sera suspendue car cela reviendrait à assigner à **X** la valeur 'val1'. Il en sera de même pour la clause (2). L'unification de la tête de clause (3) avec le sous-objectif **r2(X)** de l'appel réussira sans problème car il n'y a aucune exportation de liaison à l'appel. Le processus correspondant à la résolution de ce sous-objectif va instancier la variable **X** à la valeur 'val1' grâce au prédicat 'Y=val1'. L'unification de la tête de la clause (1) pourra reprendre, et réussira car la valeur de la variable **X** est la même que celle de l'argument de la tête de clauses. Par contre, l'unification de la tête de la clause (2) reprendra aussi, mais échouera car les deux valeurs 'val1' et 'val2' ne sont pas unifiables.

Puisqu'une unification qui a été suspendue peut reprendre plus tard, l'unification correspond à une activité continue. Elle ne se termine que sur une réussite ou un échec définitif.

c) Evaluation de la garde

L'évaluation de la garde peut être réalisée en parallèle avec l'unification de l'appel et de la tête de clause.

Les différents sous-objectifs formant la garde sont résolus en parallèle-ET. Les processus concurrents correspondant à ces sous-objectifs sont synchronisés grâce à l'instanciation des variables qui leur sont communes. Un processus consommateur d'une variable sera suspendu jusqu'au moment où le producteur de cette variable l'aura liée à un terme non variable. On remarque, dans l'exemple donné au point b, que le processus **r1** est suspendu en attendant que **r2** ait donné une valeur à **X**.

Si plusieurs processus producteurs existent pour la même variable partagée et que les valeurs qu'ils veulent assigner à la variable sont incompatibles, seul le premier qui instanciera la variable réussira. Puisque les instanciations que les autres auraient voulu effectuer se sont transformées en tests (car la variable est devenue un terme non variable), ces tests échoueront si les valeurs ne sont pas les mêmes.

Exemple : appel : **r1(X), r2(X),r3(X).**

la conjonction de l'appel correspond à la garde d'une clause évaluée pour un autre appel r

(r :- r1(Z), r2(Z), r3(Z) | ...).

clauses : **r1(Y) :- true | Y = un.** (1)
r2(Y) :- true | Y = deux. (2)
r3(Y) :- true | Y = trois. (3)

Si le processus correspondant à la résolution du sous-objectif **r2(X)** est le premier à instancier la variable commune **X**, celle-ci va recevoir la valeur 'deux'. Les deux autres processus échoueront car les tests " deux = un" et "deux = trois" sont négatifs.

Si durant sa résolution, un des sous-objectifs de la garde lie (de façon directe ou indirecte) une variable apparaissant dans l'appel avec un terme non variable ou une autre variable de l'appel, sa résolution sera suspendue au moment où il tentera d'effectuer l'unification qui exporterait une liaison à l'appel. La suspension de résolution de ce sous-objectif prendra fin dès qu'un processus concurrent correspondant à un autre sous-objectif de l'appel aura instancié la variable de l'appel qui leur est commune.

Exemples :

1. appel : **r1(X), r2(X).**
   clauses : **r1(Y) :- Y = val1 | ...** (1)
   r2(Y) :- true | Y = val1. (2)

L'évaluation de la garde de la clause (1) est suspendue car on exporterait de façon directe la valeur 'val1' à la variable **X** de l'appel via la résolution du prédicat '='. Le processus concurrent correspondant au sous-objectif **r2(X)** de l'appel instancie la variable **X** sans aucun problème car l'exportation de la valeur est faite à partir du corps de la clause. L'évaluation de la garde de la clause (1) peut donc reprendre, elle n'est plus suspendue. La résolution du prédicat '=' dans la garde devient un simple test puisque la variable est maintenant instanciée.

2. Cet exemple est emprunté à [Ue85a] et décrit une situation plus délicate.

   appel : **r1(X), r2(X).**
   clauses : **r1(Y) :- r2(Y) | ...** (1)
   r2(Z) :- true | Z = ok. (2)

Pour résoudre la partie passive de la clause (1), il faut tout d'abord unifier **X** et **Y**, puis résoudre **r2(Y)**. Pour résoudre **r2(Y)**, on essaie en utilisant la clause (2) d'unifier **Y** avec 'ok'. Cette unification ne peut pas instancier **X** car elle est indirectement invoquée dans la partie passive de la clause (1). Elle est donc suspendue.

Le processus correspondant au sous-objectif **r2(X)** peut se dérouler sans aucun problème car il n'exporte rien à l'appel à partir de la partie passive d'une clause. La variable **X** reçoit la valeur 'ok'.

L'unification suspendue dans le cadre de la résolution de **r2(Y)** peut reprendre car, puisque la variable **X** est instanciée, on ne doit plus lui exporter de liaison, l'ancienne exportation a été transformée en un test de valeur.

Les cas de suspension concernant l'unification de la tête et l'évaluation de la garde peuvent être résumés par la première règle de suspension de GHC :

Règle de suspension (a) :

" Toute unification invoquée de façon directe ou indirecte durant l'exécution de la partie passive d'une clause (unification de la tête ou évaluation de la garde) ne peut lier une variable apparaissant dans l'appel avec :

(1) un terme non variable

ou (2) une autre variable apparaissant dans l'appel,

avant que la clause n'ait passé le "trust". "

Seule la clause qui passe le "trust" (elle est unique) pourra exporter des liaisons à l'appel. La synchronisation entre processus concurrents est donc réalisée par suspension dans l'attente d'une valeur pour une variable partagée.

La règle de suspension (a) peut être reformulée de la façon suivante :

" La partie passive d'une clause ne peut exporter de liaisons de façon directe ou indirecte à l'appel avant que la clause n'ait passé le "trust". "

Comme vu dans les deux exemples, cette règle est utilisée pour la synchronisation de processus, on peut dès lors l'appeller *'règle de synchronisation'*.

d) Exécution du "trust"

Le trust est exécuté de façon à respecter la règle de "trust" de GHC.

Règle de "trust" :

" Une clause candidate ne peut passer le "trust" que si aucune autre clause appartenant à la même procédure n'a passé le "trust" pour la résolution du même objectif. "

Cette règle de "trust" énoncée par Ueda n'apporte rien de plus à la sémantique de l'opérateur de "trust" telle qu'elle avait été définie à la section I.2.8.

e) Exécution du corps

L'exécution du corps (ou partie active) peut être réalisée en parallèle avec l'exécution de la partie passive de la clause (= unification de l'appel et de la tête de clause et évaluation de la garde). Cette exécution du corps doit malgré tout respecter la condition suivante, qui correspond à la seconde règle de suspension de GHC :

Règle de suspension (b) :

" Toute unification invoquée de façon directe ou indirecte durant l'exécution de la partie active d'une clause (résolution du corps) ne peut lier une variable apparaissant dans la partie passive de cette clause avec
(1) un terme non variable
ou (2) une autre variable apparaissant dans la partie passive
tant que la clause n'a pas passé le "trust". "

Cette règle de suspension peut être formulée différemment:

" La partie active d'une clause ne peut exporter de liaisons de façon directe ou indirecte à la partie passive de cette clause avant que la clause n'ait passé le "trust". "

Si la résolution d'un des sous-objectifs du corps de la clause a pour effet d'instancier une variable de la partie passive (tête de clause ou garde), l'unification concernée sera suspendue. La suspension prendra fin lorsque la clause aura franchi le "trust". L'implémentation est telle qu'au passage du "trust" pour une clause donnée, les suspensions des corps des autres clauses et les évaluations de ces corps se terminent.

Exemple : appel : **r1(X), r2(X).**
clauses : **r1(Y) :- true | Y = ok.** (1)
r1(Y) :- q(val) | Y = pasok. (2)
r2(Z) :- true | ... (3)

L'exécution du corps peut commencer simultanément avec l'exécution de la partie passive.

Supposons que les deux clauses (1) et (2) soient essayées en même temps et que l'exécution des corps débute avant la fin de l'évaluation des gardes des clauses. Aucune des deux clauses n'ayant encore passé le "trust", les unifications qui se trouvent dans les corps sont suspendues car elles instancieraient une variable apparaissant dans la partie passive (**Y** est un argument de la tête de clause).

Dès que le choix de la clause candidate est fait (supposons que ce soit la clause (1)), l'unification qui avait été suspendue dans le corps de la clause qui a passé le "trust" peut être reprise, la variable **X** de l'appel sera donc instanciée à la valeur de la variable **Y** de la tête de clause (= 'ok').

Il faut noter que la suspension intervenant du niveau de l'instanciation de la variable partagée **X** permet de ne pas influencer le déroulement du processus correspondant au sous-objectif **r2(X)** dans l'appel, et ce, tant que la clause utilisée pour résoudre le sous-objectif **r1(X)** n'a pas été choisie par passage du "trust".

Dès que la clause (1) a franchi le "trust", la suspension de l'unification invoquée à partir du corps de la clause (2) est annulée.

Cette seconde règle de suspension garantit que la résolution du corps d'une clause qui n'a pas encore passé le "trust" n'affecte en rien la sélection d'une clause candidate, ni les processus concurrents correspondant aux autres sous-objectifs de l'appel. En effet, rien n'est exporté avant que la clause ne soit choisie. Les seules variables qui peuvent être modifiées avant que la clause n'ait passé le "trust" sont celles qui sont locales au corps de la clause.

Cette règle est donc utilisée pour séquencer les exécutions de la partie passive et de la partie active. En cas de risque d'exportation de liaison prématurée à partir de la partie active, l'exécution de celle-ci doit attendre la fin de celle de la partie passive (et le passage du "trust") avant de pouvoir continuer. On l'appelle la *'règle de séquencement'*.

I.5.3. Exemples de programmes GHC

1) Fusion de deux flots de données

Nous avons présenté à la section I.3.4. la spécification de ce programme, ainsi qu'une construction générale de celui-ci. Seules les caractéristiques propres à GHC seront abordées ici.

- *Construction propre à GHC*

Si les listes d'entrée X et Y ne sont pas instanciées dans l'appel, la description des arguments des têtes de clauses sous forme de listes suffit à déclencher une suspension. En effet, la règle de suspension (a) définie à la section I.5.2. prescrit automatiquement une suspension de l'unification de tête de clause où apparaissent des termes non variables avec l'appel si les arguments correspondants y sont des variables non instanciées.

Une conséquence des règles de suspension de GHC est que toute unification destinée à exporter des liaisons à l'appel doit être spécifiée dans la partie active de la clause en utilisant le prédicat prédéfini '=' qui unifie ses deux arguments. L'assignation d'une valeur à la liste **Z** doit donc être explicitement spécifiée dans le corps des clauses de la procédure. Grâce à la règle de suspension (b), aucune liaison ne sera ainsi exportée tant que la clause considérée ne sera pas choisie par passage du "trust", le non-déterminisme ne crée donc aucun risque d'inconsistance.

• *Code du programme*

```
(1) merge([Xs|X],Y,Z) :- true | Z = [Xs|W], merge(X,Y,W).
(2) merge(X,[Ys|Y],Z) :- true | Z = [Ys|W], merge(X,Y,W).
(3) merge(X,[],Z) :- true | Z = X.
(4) merge([],Y,Z) :- true | Z = Y.
```

2) Système de réservation aérienne

La spécification et la construction générale de ce problème ont été données dans le le cadre du langage Parlog, à la section I.3.4..

• *Construction propre à GHC*

La suspension nécessaire en cas de non instanciation des deux arguments de la procédure principale database est gérée par application de la règle de suspension (a) (cfr point c de la section I.5.2.) : l'unification de tête de clause sera suspendue pour les trois clauses tant que l'argument correspondant dans l'appel restera non instancié. Il faut pour cela que les arguments des têtes de clauses soient sous la forme de listes (= termes non variables). Dans ce cas, l'unification de l'argument non instancié de l'appel avec une quelconque des trois têtes de clauses serait suspendue car son résultat serait d'exporter une liaison à l'appel. La suspension sera annulée dès que le flot de commandes sera instancié.

Tout comme pour Parlog et Concurrent Prolog, la synchronisation entre les processus correspondant aux trois sous-objectifs de la procédure reserve se fait par mise à jour de variables partagées. Les suspensions nécessaires quand le producteur n'a pas encore instancié la variable partagée sont prescrites grâce à l'appel (direct ou indirect) au prédicat novar(X) dans les gardes des clauses des procédures consommateurs. Le prédicat novar(X) définit une suspension si X est une variable non instanciée. La suspension est levée dès que X a reçu une valeur. Le prédicat novar(X) ne doit pas être considéré comme faisant partie de GHC. Il fait partie en tant que tel de la primitive d'unification nécessaire pour la résolution d'un programme écrit en GHC (similaire à data(X) pour Parlog et wait(X) pour Concurrent Prolog).

Nous avons vu qu'il est interdit d'exporter une liaison à un argument de l'appel tant que le choix de la clause utilisée pour résoudre cet appel n'est pas fixé définitivement par passage du "trust". Pour ce qui est de la procédure response, les unifications pour les deux variables à instancier seront dès lors explicites (emploi du prédicat prédéfini '=' du Prolog séquentiel) et se trouveront dans la partie active des clauses. Rien ne sera donc exporté à l'appel avant qu'une des deux clauses ne passe le "trust".

• *Code du programme*

```
(1) database([info(Flight,Seats)|S],DB) :- true |
                    value(DB,Flight,Seats),
                    database(S,DB).
(2) database([reserve(Flight,Seats,Response)|S],DB) :-
                    reserve(Flight,Seats,DB,Response,DB1) |
                    database(S,DB1).
(3) database([],_) :- true | true.

(1) reserve(Flight,Seats,DB,Response,DB1) :- true |
                    value(DB,Flight,Freeseats),
                    plus(Freeseats,Seats,Leftseats),
                    respond(DB,Leftseats,Flight,Response,DB1).

(1) respond(DB,Seats,Flight,Response,DB1) :- le(0,Seats) |
                    modify(DB,Flight,Seats,DB1),
                    Response = true.
(2) respond(DB,Seats,Flight,Response,DB1) :- lt(Seats,0) |
                   DB1 = DB,
                   Response = false.

      le(X,Y) :- novar(X), novar(Y), !, X < Y.
      le(X,Y) :- novar(X), novar(Y), !, X = Y.
      lt(X,Y) :- novar(X), novar(Y), !, X < Y.
      plus(X,Y,Z) :- novar(X), novar(Y), !, Z is X - Y.
```

I.6. Comparaison des trois langages

Différents critères vont être analysés : puissance d'expression, degré de parallélisme effectif, complexité de l'implémentation, facilité de construction de programmes, clarté du code des programmes et possibilité d'établir qu'un programme est correct.

Même si aucun des trois langages ne se détache distinctement des autres sur base des critères énoncés, cette comparaison permettra de faire ressortir les principales différences existant entre Parlog, Concurrent Prolog et GHC.

I.6.1. Puissance d'expression

La puissance d'expression d'un langage peut être définie par l'étendue de son domaine d'application, par les possibilités qu'il offre.

Par rapport à Concurrent Prolog et à GHC, Parlog offre la possibilité supplémentaire d'obtenir un ensemble de solutions, plutôt qu'une solution unique. Nous avons néanmoins décidé à la section I.3.1. de ne pas aborder cette caractéristique de Parlog, afin de ne comparer que ce qui est comparable. Il faut toutefois souligner le fait que cette possibilité donne à Parlog une supériorité certaine par rapport aux deux autres langages.

Pour ce qui est de leur puissance d'expression, Concurrent Prolog, GHC et la partie de Parlog concernant les procédures à une solution peuvent probablement être jugés équivalents. Ils permettent tous d'écrire à peu près le même ensemble de programmes correspondant à des problèmes caractéristiques de programmation parallèle.

I.6.2. Degré de parallélisme effectif

Trois parallélismes différents vont être considérés : le parallélisme tête-garde-corps évoqué dans le cadre de GHC (cfr section I.5.2.), le parallélisme-OU et le parallélisme-ET.

a) parallélisme tête-garde-corps

Ce parallélisme peut être défini comme étant la possibilité d'effectuer de façon tout à fait simultanée les trois actions suivantes : unifier une tête de clause avec l'appel, évaluer la garde de cette même clause, et résoudre le corps de la clause. Ce parallélisme n'étant proposé que pour GHC, l'analyse de la façon dont il est réduit ne pourrait nous être d'aucune utilité pour la comparaison des trois langages.

b) parallélisme-OU

L'évaluation en parallèle-OU des différentes clauses candidates doit être contrôlée de façon à ne pas créer d'interférence en exportant des liaisons à l'appel. Ce contrôle de l'évaluation implique une réduction du parallélisme, mais à différents degrés selon les langages.

Comme expliqué dans [Vla86], le degré de parallélisme d'un programme dépend fort du degré d'interférence entre ses composants. En réduisant les interférences, on peut donc augmenter le parallélisme. Pour les trois langages considérés, nous allons voir comment l'élimination plus ou moins complète des risques d'interférence permet une évaluation complètement parallèle-OU des différentes clauses.

Concurrent Prolog ne réduit pas le parallélisme-OU car la présence d'environnements multiples lui permet de travailler sur les variables locales à chaque clause. La disparition de manipulations de variables globales entraîne l'absence de risque d'interférence. Il ne faut donc pas limiter le parallélisme en créant des sections critiques. Cette technique de transformation des variables globales en versions privées qui sont locales aux différents composants est appellée réappellation (renaming). Cela correspond bien au concept introduit dans [Vla86], car aucune synchronisation n'est requise entre les évaluations des différentes clauses d'une même procédure pour la résolution d'un appel initial.

Parlog supprime aussi la manipulation de variables globales durant l'évaluation des clauses. Tout programme Parlog compilé est transformé sous forme standard. C'est sous cette forme qu'il sera exécuté. La transformation comporte, entre-autres, un report au début du corps de chaque clause des liaisons aux variables de type output de l'appel. Rappellons qu'il y a un contrôle à la compilation afin de vérifier que les gardes sont sûres, c'est-à-dire qu'elles ne contiennent aucun prédicat dont la résolution entraînerait une exportation de liaison à une variable de l'appel correspondant à un argument de type input. Les programmes qui passent avec succès l'étape de compilation sont donc tels que rien n'est exporté durant l'évaluation parallèle-OU, toute instanciation de variable étant reportée après le passage du "commit". Comme la clause qui passe le "commit" est unique, il n'y a pas de risque d'interférence pour la mise à jour des variables globales, des sections critiques ne sont pas nécessaires. Parlog ne réduit donc pas le parallélisme-OU.

Pour ces deux langages, les évaluations des clauses sont vraiment exécutées en parallèle-OU. Il faut cependant remarquer que les façons dont les risques d'interférence sont évités limitent le parallélisme à un autre niveau. Considérons l'exemple suivant (en Concurrent Prolog) :

```
appel : r1(Y?,X), r2(X?).
clauses : r1(A,B) :- A > 5, B = true | ...      (1)
          r1(A,B) :- A < 5, B = false | ...     (2)
          r1(A,B) :- A = 5, B = ok | ...        (3)
```

Les évaluations des trois clauses se font en parallèle car des variables locales leur sont attribuées. Cependant, le processus correspondant à la résolution du sous-

objectif **r2** est suspendu en attente d'une valeur pour la variable partagée **X**. Cette variable est globale et ne recevra une valeur que quand la clause sélectionnée sera désignée par passage du "commit". C'est seulement à ce moment (tout comme pour Parlog) que les liaisons output sont effectuées aux variables de l'appel. On peut dès lors remarquer que des gardes dont l'évaluation prend beaucoup de temps (en raison du nombre de prédicats ou de la complexité de ceux-ci) limitent le parallélisme-ET, car les processus consommateurs concurrents sont suspendus longtemps en attente d'une valeur pour une variable partagée.

Les caractéristiques de Concurrent Prolog et Parlog en ce qui concerne le degré de parallélisme-OU effectif sont donc semblables. Voyons ce qu'il en est pour GHC.

Un programme *GHC* peut être comparé à un programme Parlog sous forme standard. En effet, aucune exportation de liaison à l'appel n'a lieu tant que la clause n'a pas passé le "commit". Il n'y a donc pas de gestion d'environnements multiples. Les exportations de liaisons seront spécifiées explicitement dans le corps. Prenons un exemple afin d'illustrer cette similitude entre GHC et la forme standard de Parlog.

- ◇ programme Parlog :
  mode r1(^).
  r1(un) :- ...
  r1(deux) :- ... •
- ◇ forme standard du programme Parlog :
  r1(X) :- **true | X** := **un** ...
  r1(X) :- **true | X** := **deux** ...
- ◇ programme GHC :
  r1(X) :- **true | X** = **un** ...
  r1(X) :- **true | X** = **deux** ...

Un programme en GHC permet donc d'obtenir un degré de parallélisme-OU similaire à celui atteint par le même programme en Parlog ou en Concurrent Prolog.

c) Parallélisme-ET

Pour ce qui est de la résolution en parallèle-ET des différents sous-objectifs d'un appel, la synchronisation entre les processus correspondant aux sous-objectifs s'effectue de la même façon pour les trois langages : les processus concurrents communiquent par mise-à-jour de variables partagées et sont suspendus s'ils doivent attendre l'exportation d'une valeur par un processus producteur pour une variable commune. Ces suspensions sont spécifiées en Parlog par le mode associé à la relation, en Concurrent Prolog par des annotations "read-only" associées à des variables de l'appel, en GHC par la règle de suspension (a).

La durée des suspensions est semblable pour les trois langages : lorsqu'une variable partagée apparaissant dans l'appel n'est pas instanciée lors de l'évaluation d'une clause pour un processus consommateur de cette variable, l'évaluation est suspendue. Elle reprend dès que le processus producteur pour cette variable lui a assigné une valeur.

d) Conclusion

Si les applications GHC sont bien rédigées, de façon à ne pas entraîner de suspensions inutiles, on peut dire que le degré de parallélisme effectif est semblable pour les trois langages.

I.6.3. Complexité de l'implémentation

Il est intéressant de considérer la complexité de l'implémentation des différents langages. L'implémentation de Parlog exige la compilation des programmes et leur transformation sous forme standard, ainsi qu'une série de primitives d'unification spécialisées sont nécessaires. Pour ce qui est de Concurrent Prolog, il faut implémenter toute une gestion d'environnements multiples. Quant à GHC, il ne nécessite qu'une primitive d'unification unique et pas d'environnements multiples. Cette facilité d'implémentation est dûe au fait que les programmes GHC peuvent être comparés à des programmes Parlog déjà transformés en forme standard.

Pour ce qui est de ce critère, GHC se détache donc favorablement par rapport aux deux autres langages.

I.6.4. Facilité de construction de programmes

La facilité à construire un programme signifie que la démarche que le programmeur doit effectuer pour traduire les spécifications de son application en un programme dans le langage concerné soit la plus simple possible. Cette facilité doit aussi être présente pour ce qui est de se convaincre que le programme est correct. Le langage le plus simple à utiliser serait un langage pour lequel il y aurait une exacte correspondance entre les concepts énoncés dans les spécifications et les constructions offertes par le langage.

Une *première approximation* de la facilité à construire des programmes en Parlog, Concurrent Prolog et GHC peut être réalisée par une *comparaison des sémantiques procédurales* de ces trois langages. On peut en effet remarquer que plus la sémantique procédurale d'un langage est complexe, moins la construction de programmes est aisée.

Nous ne reprendrons que quelques points sur lesquels les trois sémantiques procédurales diffèrent.

La présence du parallélisme-OU nécessite en Parlog la définition de mode output pour certains arguments. Ceux-ci ne seront pas instanciés avant que le choix de la clause candidate ne soit effectué. L'effet de l'application de ce mode output n'est pas aisé à comprendre car il faut pour cela considérer l'étape de compilation du programme Parlog. Concurrent Prolog ne nécessite aucune construction spéciale pour éviter des instanciations prématurées risque dû au parallélisme-OU) car la gestion d'environnements multiples évite tout problème. De façon à ce que le parallélisme-OU n'engendre aucune inconsistance, il faut interdire des exportations de liaisons à l'appel avant que la clause ne soit choisie. GHC implémente cette interdiction par l'application de la règle de suspension (a).

L'existence d'un parallélisme-ET nécessite une synchronisation entre processus. Le moyen grâce auquel la synchronisation entre processus est effectuée en Parlog est l'utilisation d'un mode input pour l'argument commun aux processus devant se synchroniser. En Concurrent Prolog, des annotations "read-only" sont employées et en GHC, c'est la règle de suspension (a) qui est appliquée. On voit donc que cette règle de suspension (a) a deux fonctions : éviter les problèmes issus du parallélisme-OU et synchroniser les processus pour le parallélisme-ET.

Le langage GHC offre la possibilité supplémentaire d'un parallélisme entre l'unification d'une tête de clause, l'évaluation de sa garde, et l'exécution de son corps (parallélisme tête-garde-corps). Ce parallélisme peut aussi être à l'origine de problèmes si une exportation de liaison est faite durant l'exécution du corps et ce, avant que la clause ne soit sélectionnée par passage du "trust". C'est pourquoi la seconde règle de suspension (b) a été définie. Cette possibilité de parallélisme supplémentaire complique la sémantique procédurale de GHC par rapport à celles de Parlog et Concurrent Prolog.

Nous pouvons dire en conclusion de cette première approximation que les sémantiques procédurales de ces trois langages sont fort complexes. Un travail considérable serait nécessaire pour définir ces sémantiques de façon claire et complète. Pour ce qui est de la comparaison entre les trois langages, Concurrent Prolog semble avoir la sémantique la moins compliquée. La complexité de celle de Parlog est plus élevée à cause de la définition des modes output. La présence d'un parallélisme tête-garde-corps pour GHC augmente le degré de complexité de sa sémantique procédurale.

Une *seconde approche* de la facilité de construction de programmes peut être effectuée en *comparant la construction du programme de réservation aérienne*, dans sa partie propre à chacun des trois langages (cfr sections I.3.4.,

I.4.3., I.5.3.). On peut constater que la nécessité de définir des modes output rend le processus de construction propre à Parlog plus compliqué que celui propre à Concurrent Prolog. La nécessité pour GHC de considérer les règles de suspension et d'écrire de façon explicite les unifications dans le corps des clauses constitue aussi une contrainte supplémentaire par rapport à Concurrent Prolog. C'est donc Concurrent Prolog qui se détache favorablement.

Cette conclusion est identique à celle issue de l'analyse des sémantiques procédurales car les constructions propres à chaque langage mettent en évidence les différences existant au niveau de la sémantique procédurale de chacun de ceux-ci. Il était donc normal que le résultat obtenu ci-dessus soit confirmé.

I.6.5. Clarté du code des programmes

Un programme clair est un programme pour lequel on peut à la lecture imaginer ce qui se passe à l'exécution sans trop de difficultés. Le code d'un tel programme ne doit comporter aucune construction compliquée qui rendrait sa compréhension plus difficile. Le texte peut donc être plus long en nombre de caractères, pour autant que le contenu en soit plus simple.

Les programmes GHC semblent être plus clairs que les programmes Parlog car, pour GHC, les unifications qui exportent des liaisons sont placées dans le corps des clauses. On ne doit donc pas, comme pour Parlog, se demander comment se passe la transformation à la compilation pour savoir quand les exportations auront effectivement lieu.

Il peut être important pour la clarté d'un programme logique parallèle de pouvoir déterminer à la lecture de la définition d'une procédure le rôle que cette procédure sera amenée à jouer : consommateur ou producteur d'une valeur pour une variable. Voyons si ce rôle peut être déterminé de façon semblable pour les trois langages. Le même programme servira d'illustration en Parlog, en Concurrent Prolog et en GHC.

1. programme en *Parlog*

```
appel : r1(X), r2(X).
clauses :   mode r1(?).
            r1(true) :- ...      (1)
            r1(false) :- ...     (2)

            mode r2(^).
            r2(X) :- X = true ...  (3)
            r2(X) :- X = false ... (4)
```

L'argument **X** de l'appel **r1** doit toujours être instancié, car sinon, le mode input impose une suspension.

2. programme en *Concurrent Prolog*

appel : **r1(X?), r2(X).**
clauses : **r1(true)** :- ... (1)
r1(false) :- ... (2)

r2(X) :- **X = true** ... (3)
r2(X) :- **X = false** ... (4)

Avec l'appel **r1(X?)**, il est nécessaire que X soit instancié. L'unification de l'appel et de la tête de clause revient donc à faire un test de la valeur de la variable.

Si l'appel est **r1(X)**, l'argument **X** de **r1** ne doit pas nécessairement être instancié, l'utilisation de **r1** a alors comme conséquence de donner une valeur à **X**.

r1 peut donc être employé pour tester une valeur en entrée ou pour exporter une valeur, selon la façon dont l'appel est écrit.

3. programme en *GHC*

appel : **r1(X), r2(X).**
clauses : **r1(true)** :- ... (1)
r1(false) :- ... (2)

r2(X) :- **true | X = true** ... (3)
r2(X) :- **true | X = false** ... (4)

L'argument **X** de **r1** doit toujours être instancié, conformément à la règle de suspension (a) car sinon, cela reviendrait à exporter une valeur à partir de la partie passive d'une clause, ce qui est sanctionné par une suspension. Une unification passive (par unification de la tête de clause) détermine donc un mode input. Pour **r2**, l'unification active via l'utilisation du prédicat ' = ' dans le corps correspond à un mode output pour l'argument considéré.

Sur base uniquement de la définition des procédures **r1**, on peut déterminer que les procédures **r1** de GHC et de Parlog seront toujours des consommateurs de la variable **X**, quel que soit l'appel pour lequel elles sont évaluées. La procédure **r1** de Concurrent Prolog peut être soit consommateur, soit producteur pour la variable **X**, selon l'appel d'origine.

Pour ce qui est des procédures **r2**, il n'est pas nécessaire de connaître l'appel car leur seule définition suffit pour affirmer qu'elles sont producteurs.

GHC et Parlog semblent être plus clairs pour cette caractéristique.

Cependant, s'il faut définir une procédure pouvant tour à tour être consommateur ou producteur pour la même variable, il faut en écrire deux versions pour Parlog et GHC, tandis qu'une seule version suffira pour Concurrent Prolog. La possibilité offerte par Concurrent Prolog est un avantage car il n'y aura qu'une seule version de la même procédure, quel que soit son emploi (clarté des programmes plus grande car on n'ajoute pas de procédures inutiles).

Au niveau de la syntaxe, on peut remarquer que, malgré la lourdeur pour GHC de représentation d'une garde ou d'un corps vide par le prédicat 'true', c'est la syntaxe du langage GHC qui est la plus simple : elle ne comporte aucune construction spéciale pour contrôler le parallélisme. Parlog, quant à lui, doit définir le mode pour chaque relation et Concurrent Prolog utilise les annotations "read-only".

En considérant l'ensemble des observations faites en ce qui concerne la clarté des programmes, on peut en déduire que le code obtenu en Concurrent Prolog est plus clair que celui des deux autres langages.

I.6.6. Possibilité d'établir qu'un programme est correct

La *validité totale* d'un programme implique que soient vérifiées les propriétés de validité partielle (le programme réalise sa spécification) et de terminaison (l'exécution du programme ne comporte pas de boucle infinie). La notion de validité d'un programme sera abordée plus en détails à la section IV.1., dans le cadre de la méthodologie de conception d'applications robotiques en GHC.

La *validité partielle* est d'autant plus difficile à vérifier que le processus de se convaincre que le programme réalise bien sa spécification est complexe. Cette difficulté est directement liée à la complexité de la sémantique procédurale du langage considéré. Ce problème ayant déjà été abordé à la section I.6.4., nous ne pouvons qu'en répéter la conclusion : la haute complexité des sémantiques procédurales des trois langages logiques parallèles considérés réduit les chances de prouvabilité des programmes résultants.

En ce qui concerne la *terminaison*, nous avons vu à la section I.1.8. que les deux principaux problèmes pouvant survenir sont l'interblocage et le blocage individuel permanent. Les processus en compétition pour des ressources sont ici les processus de résolution d'objectifs, et les ressources sont les accès aux variables partagées. Les deux risques peuvent se présenter.

L'*interblocage* est le cas où un processus **P1** est suspendu dans l'attente d'une valeur pour une variable **X**. Sa suspension l'empêche de produire une valeur pour la variable **Y**. Or, le processus **P2** producteur de la variable **X** est suspendu en attente

d'une valeur pour **Y**. S'il n'y a qu'un producteur pour chacune des deux variables, les deux processus **P1** et **P2** resteront interbloqués.

Le *blocage individuel permanent* est la situation où un processus **P1** est suspendu en attente d'une valeur pour une variable **X**. S'il n'y a aucun processus concurrent producteur pour cette variable, le processus **P1** restera suspendu indéfiniment.

Ces deux problèmes se poseront toujours, quel que soit le langage considéré. Cependant, certains langages peuvent induire plus de risques que d'autres. Ces risques sont liés aux différentes primitives que les langages offrent. Une comparaison des trois langages sur ce point peut être réalisée en considérant la multiplicité des types de suspension prescrits dans la sémantique procédurale de chacun d'eux. En effet, un nombre de suspensions élevé accroît les risques de non-terminaison d'un programme. Le langage pour lequel le plus de types de suspension existent offre donc le plus de risques de non-terminaison.

Pour ce qui est de Parlog, un seul type de suspension est cité : cette suspension est prescrite en cas de non respect du mode input pour un argument (voir section I.3.2., et point b de la section I.3.3.)

En Concurrent Prolog, la seule situation où une suspension est envisagée est l'unification d'une variable non instanciée annotée "read-only" avec un terme non variable (voir section I.4.1.).

Le langage GHC, quant à lui, prescrit deux types de suspension. Ces suspensions sont liées aux règles de suspension (a) et (b) énoncées aux points c et e de la section I.5.2. Elles correspondent respectivement aux cas où il y a un risque d'exporter une liaison à l'appel lors de l'évaluation d'une clause (unification de la tête de clause et évaluation de la garde), et un risque de lier une variable de la partie passive (tête ou garde) lors de l'exécution de la partie active d'une clause (résolution du corps). Le premier cas de suspension de GHC est semblable aux suspensions prescrites pour Parlog et pour Concurrent Prolog : ces suspensions sont utilisées pour synchroniser les processus. Rien de semblable au second cas de suspension de GHC n'est présent dans les sémantiques procédurales de Parlog et de Concurrent Prolog. Mais cette particularité de GHC étant dûe au parallélisme tête-garde-corps et ce parallélisme n'étant pas envisagé pour les deux autres langages, il est normal que Parlog et Concurrent Prolog ne reprennent pas ce cas de suspension. GHC semble donc induire plus de risques quant à la non-terminaison de programmes car deux types de suspension doivent être envisagés. Il faut cependant remarquer que si GHC n'offrait pas un parallélisme supplémentaire par rapport aux deux autres langages, les risques de non-terminaison liés à l'utilisation de GHC, Parlog et Concurrent Prolog seraient semblables.

I.6.7. Conclusion

Cette analyse ne permet pas de choisir un langage car les résultats ne sont pas assez tranchés : chaque langage a ses avantages et ses inconvénients. Notons

toutefois que c'est le langage Concurrent Prolog qui semble présenter le moins d'inconvénients.

Beaucoup d'études ayant déjà été réalisées pour les langages Parlog et Concurrent Prolog, il était intéressant de choisir le langage GHC pour la rédaction d'applications robotiques. Il serait ainsi possible de faire une analyse concrète des possibilités offertes par ce langage, sans pour autant reprendre des sujets déjà abordés dans des publications antérieures.

Afin de pouvoir utiliser GHC et tester le plus rapidement possible son adéquation aux applications de robotique, il faudrait disposer d'un compilateur ou d'un interpréteur. C'est un interpréteur qui sera tout d'abord écrit. Sa description fait l'objet du chapitre suivant. On pourra, par la suite, utiliser un compilateur afin d'améliorer les performances des programmes rédigés en GHC.

Chapitre II

Développement d'un interpréteur du langage Guarded Horn Clauses

II.1. Introduction

Comme aucun interpréteur pour GHC n'a, semble-t-il, jamais été publié, c'est sur base des travaux réalisés par Shapiro pour Concurrent Prolog dans [Sh83] que l'interpréteur de GHC a été écrit.

L'implémentation du langage GHC tel qu'il a été défini à la section I.5. n'utilise pas du parallélisme effectif car ce parallélisme n'est pas possible sur les architectures dont nous disposons pour le moment. Tout ce qui doit normalement être exécuté en parallèle sera donc fait de façon séquentielle, tout en simulant un certain non-déterminisme. Cette restriction peut être considérée comme valable car l'exécution séquentielle est un cas particulier de l'exécution parallèle.

II.2. Restrictions par rapport au GHC pur

On peut distinguer trois parallélismes dans la sémantique procédurale du GHC pur :

1. Le parallélisme-ET, auquel est associé un non-déterminisme-ET au niveau de la résolution des sous-objectifs d'une conjonction.

2. Le parallélisme-OU, auquel est associé un non-déterminisme-OU au niveau de la sélection de la clause qui est utilisée pour réduire un objectif.

3. Le parallélisme entre l'exécution de l'unification de l'appel et d'une tête de clause, l'évaluation de la garde de cette même clause, et l'exécution de son corps (appellé le parallélisme tête-garde-corps).

Voyons maintenant quelles sont les restrictions apportées au niveau de ces trois types de parallélisme dans l'implémentation proposée.

II.2.1. Parallélisme-ET

Le parallélisme-ET est simulé par une forme restreinte du non-déterminisme-ET. La restriction porte sur le fait qu'à chaque exécution du même programme, l'ordre dans lequel les sous-objectifs d'une conjonction sont résolus sera identique. Il est néanmoins "non-déterministe-ET" car, sur base de la place des objectifs dans une conjonction, on ne sait pas dire quel sera l'objectif suivant qui sera choisi pour être résolu.

De plus amples détails à ce sujet seront donnés dans la description de l'interpréteur (voir le point a de la section II.5.3.).

II.2.2. Parallélisme-OU

Le parallélisme-OU n'est pas simulé : il n'y a aucun non-déterminisme-OU dans l'exécution d'un programme. L'ordre dans lequel les clauses d'une procédure sont essayées pour la résolution d'un appel donné est l'ordre dans lequel elles apparaissent dans le code de la procédure. Il n'y a donc pas de choix aléatoire d'une clause parmi les clauses d'une procédure.

La simulation d'un réel non-déterminisme-OU n'a pas été implémentée dans l'interpréteur lui-même car il nous a semblé plus intéressant d'étudier le mécanisme d'unification propre à GHC. On pouvait en effet réaliser une comparaison entre GHC et Concurrent Prolog en ce qui concerne l'unification, tandis que pour le parallélisme-OU, aucune comparaison n'était possible car celui-ci n'est pas non plus simulé dans l'interpréteur écrit par Shapiro pour Concurrent Prolog (cfr [Sh83]).

Les clauses d'une procédure sont donc évaluées séquentiellement de la façon suivante : si l'évaluation d'une clause se termine sur un échec (cette clause est non-candidate), on passe à l'évaluation de la clause suivante. De même, si l'unification de l'appel et de la tête d'une clause résulte en une exportation de liaison à l'appel, il faut suspendre l'évaluation de la clause en passant à l'évaluation de la clause suivante. *On ne fait donc aucune distinction entre échec et suspension.* La résolution d'un objectif pour lequel il n'y a pas de clause immédiatement candidate sera reportée, que la cause en soit un échec définitif ou une simple suspension. En effet, un tel objectif étant considéré comme un objectif non encore résolu, on effectuera un essai ultérieur de résolution comme si aucun essai préalable n'avait déjà été réalisé. Nous verrons cependant dans la suite que cette absence de distinction entre échec et suspension est à l'origine de certains problèmes.

Par absence de tout parallélisme-OU, on n'évaluera jamais les gardes de plusieurs clauses de façon simultanée. Il n'y a donc plus de risque d'interférence dû à la manipulation simultanée de variables de l'appel par plusieurs gardes en cours

d'évaluation car si une seule garde est évaluée à la fois, on peut considérer les évaluations des différentes gardes d'une même procédure comme étant tout à fait indépendantes. Cette limitation supprime des nécessités de suspension car, de par la façon dont la clause utilisée pour la résolution d'un appel est choisie, la règle de "trust" (cfr point d de la section I.5.2.) sera toujours respectée. En effet, la clause qui franchit le "trust" est toujours unique puisqu'il n'y a jamais plusieurs clauses essayées en parallèle-OU.

II.2.3. Parallélisme tête-garde-corps

Ce parallélisme n'est pas du tout simulé, la stratégie adoptée est la suivante.

(1) On unifie l'appel et la tête d'une clause.
(2.1) Si l'unification aboutit alors
on évalue la garde de la clause considérée
(2.1.1) Si la garde est satisfaite
alors on réduit l'appel au corps de la clause
(2.1.2) sinon on se retrouve dans la situation (1), avec la clause suivante comme nouvelle clause.
(2.2) sinon, on se retrouve dans la situation (1) avec la clause suivante comme nouvelle clause.

Rappellons que l'unification aboutit si et seulement si les termes sont unifiables et aucune liaison n'est exportée à l'appel, et que la garde est satisfaite si et seulement si tous ses sous-objectifs ont été résolus avec succès.

De cette stratégie résulte le fait que l'exécution de la partie active d'une clause n'aura lieu qu'après le succès de la garde. La partie active d'une clause n'exportera donc jamais de liaisons à la partie passive de cette clause avant que la clause ne passe le "trust". La seconde règle de suspension (b) (cfr point e de la section I.5.2.) sera ainsi toujours vérifiée.

Pour ce qui est de l'évaluation de la garde, l'utilisation de l'environnement du Prolog séquentiel pour l'interpréteur permet de disposer du mécanisme de rétroparcours. Ce mécanisme défait automatiquement les liaisons effectuées durant la résolution d'une conjonction d'objectifs si cette résolution échoue. Il n'y a donc pas de problème s'il est nécessaire d'évaluer plusieurs clauses avant de trouver une clause candidate : les liaisons sont chaque fois annulées.

Cependant, on ne peut permettre que la garde exporte des liaisons car cela irait à l'encontre de la règle de suspension (a) (cfr point c, section I.5.2.). Il faudrait dès lors suspendre la résolution car on exporterait une liaison avant de franchir le "trust".

Exemple de situation qu'il faudrait pouvoir détecter :

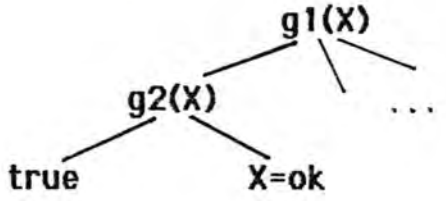
appel : **h(Y).**
clauses : **h(X) :- g1(X) | ...** (1)
g1(X) :- g2(X) | ... (2)
g2(X) :- true | X = ok. (3)

Si aucune vérification n'est effectuée, la variable **Y** apparaissant dans l'appel sera instanciée dans la garde via le prédicat **g1**.

Une méthode à appliquer pour effectuer une telle vérification serait de parcourir l'arbre de résolution ET/OU (voir ci-dessous) jusqu'au moment où on trouve une instanciation de la variable concernée ou, s'il n'existe pas d'instanciation pour la variable, jusqu'au moment où les feuilles de l'arbre sont atteintes.

Si les feuilles de l'arbre sont atteintes sans détecter d'instanciation, on peut essayer de résoudre le prédicat concerné (ici g1(X));

sinon, la résolution du prédicat doit être suspendue en attendant une instanciation qui serait effectuée pour la même variable, mais par un autre processus.



La complexité des tests qu'il faudrait réaliser pour implémenter la détection de telles situations dans l'interpréteur est assez grande. On va donc interdire à l'utilisateur d'écrire des gardes contenant des prédicats effectuant directement ou indirectement des liaisons à l'appel.

Cette contrainte est assez lourde car la vérification manuelle par le programmeur risque de ne pas être simple. Cette contrainte peut cependant être comparée à celle qui est faite pour le Flat Concurrent Prolog (cfr [Mi84]) où on ne peut utiliser dans la garde que des prédicats prédéfinis (les prédicats de l'utilisateur sont interdits). On pourrait dès lors parler ici d'implémentation d'un Flat GHC.

II.2.4. Simulation des suspensions

Il y a deux cas où il faut simuler une suspension pour éviter de violer la règle de suspension (a). : une exportation à l'appel à partir de la tête de la clause et une exportation à l'appel à partir de la garde de la clause.

Comme nous l'avons vu à la section II.2.3., l'exportation d'une liaison à l'appel *à partir de la garde* n'est pas sanctionnée par une suspension. Dans ce cas, la

suspension n'est pas simulée à cause de la complexité de détection d'une telle situation, qu'on exclut dès lors.

La suspension dûe à une exportation illicite vers l'appel *à partir de la tête de clause* sera prescrite au niveau de la primitive d'unification de l'interpréteur de GHC.

En effet, si un tel cas se présente, la résolution de la primitive d'unification avec pour arguments l'appel et la tête de clause échouera. Après cet échec, la clause suivante de la même procédure sera essayée. Si l'unification n'est possible avec aucune des têtes de clauses de la procédure, l'appel échouera (pour simuler une suspension) et sera à nouveau considéré comme un objectif non encore résolu. Il sera réessayé lorsqu'il sera à nouveau sélectionné en fonction de la stratégie de résolution adoptée.

Cette simulation de suspension prendra fin lorsque :

- soit l'unification de l'appel avec une autre tête de clause aura réussi,
- soit l'appel à résoudre aura définitivement échoué (suite à la détection d'une situation d'échec cyclique, cfr point c de la section II.5.3.),
- soit un nouvel essai d'unification de l'appel avec cette même clause aura réussi.

Cette simulation n'est pas tout-à-fait fidèle en ce sens que l'appel à résoudre ne sera pas nécessairement réessayé dès que les risques d'exportation de liaison à l'appel auront disparu (c'est-à-dire dès que la résolution d'un autre appel aura instancié la(les) variable(s) qui rendai(en)t l'unification impossible). L'ordre de sélection des objectifs non encore résolus étant fixé par une stratégie équitable (cfr point a de la section II.5.3.), il est certain que l'appel suspendu sera réveillé dans un délai fini. La reprise après la suspension sera donc peut-être retardée, mais aura certainement lieu.

II.2.5. Justification des restrictions

Les restrictions du parallélisme telles qu'elles viennent d'être décrites préservent la validité d'un programme rédigé en GHC. En effet, tout programme GHC correct donnera, avec l'interpréteur proposé ici, un résultat appartenant à l'ensemble des résultats corrects susceptibles d'être obtenus avec une architecture parallèle (c'est-à-dire avec une implémentation sans aucune restriction).

Cette affirmation peut être justifiée par le fait que toutes les exécutions possibles d'un programme GHC avec les restrictions de parallélisme mentionnées ci-dessus (GHC-restreint) sont strictement incluses dans l'ensemble des exécutions sans aucune restriction de parallélisme du même programme (GHC-pur).

En effet, au niveau du *parallélisme-OU*, l'absence de tout non-déterminisme dans l'implémentation proposée ici a pour conséquence que pour toute exécution du même programme GHC, l'ordre dans lequel les clauses d'une procédure sont essayées

sera identique. Cet ordre est un élément de l'ensemble des ordres envisageables avec un non-déterminisme-OU.
Cet ordre permet d'affirmer que la règle de "trust" telle qu'elle a été énoncée au point d de la section I.5.3. sera toujours vérifiée.

En ce qui concerne le *parallélisme tête-garde-corps*, nous avons vu qu'il n'était pas simulé. L'exécution séquentielle telle qu'elle a été décrite correspond cependant à un des ordres d'exécution possibles dans le cas où ce parallélisme serait disponible. Cette exécution séquentielle a pour conséquence que la règle de suspension (b) sera toujours vérifiée.

L'approche du *parallélisme-ET* n'est pas vraiment non-déterministe en ce sens que l'ordre de résolution des objectifs d'un conjonction sera le même pour toute exécution. Cet ordre correspond à un des ordres qu'il serait possible d'obtenir si le parallélisme-ET était implémenté sans restriction.

La règle de suspension (a) est vérifiée grâce à la suspension simulée par la primitive d'unification de l'interpréteur et à l'interdiction d'écrire des gardes contenant des prédicats exportant directement ou indirectement des liaisons à l'appel.

On peut dès lors affirmer que l'exécution d'un programme GHC-restreint est correcte et respecte la sémantique procédurale décrite par Ueda dans [Ue85a], [Ue85b].

II.2.6. Modification de la syntaxe du GHC pur

La seule différence se situe au niveau de l'opérateur de trust. Le symbole " **|** " étant déjà utilisé comme opérateur pour la représentation de listes dans l'environnement du Prolog séquentiel (LPA MacPROLOG, syntaxe Edinburg), il fallait soit le redéfinir, soit utiliser un autre symbole. La seconde solution a été choisie, en remplaçant le " **|** " par un " **!** ".
Il n'y a aucune ambiguïté avec le symbole du "cut" du Prolog séquentiel car un "cut" ne peut apparaître dans un programme GHC.

Au niveau syntaxique, la similitude est complète entre un programme écrit en GHC et un programme écrit en Prolog séquentiel selon la syntaxe définie par Clocksin et Mellish (cfr [ClMe81]). Cela offre la possibilité de disposer des méta-prédicats prédéfinis pour le Prolog séquentiel et de les appliquer au langage GHC.

La syntaxe du langage GHC implémenté peut dès lors être définie de la façon suivante (à comparer à la section I.5.1.).

Un programme GHC est un ensemble fini de clauses gardées de la forme :

H :- G1, ..., Gm, !, B1, ..., Bn. avec m,n ≥ 0

H est la tête de clause,
Gi sont les objectifs de la garde de la clause,
Bi sont les objectifs du corps de la clause.

Les variables sont représentées par des symboles dont la première lettre est une majuscule. Les noms de fonctions et de prédicats commencent par une lettre minuscule.

Le prédicat vide 'true' est utilisé pour représenter une garde ou un corps vide (m ou n = 0).

II.3. Langage de l'interpréteur

Le langage choisi pour la rédaction de l'interpréteur GHC est le Prolog séquentiel. Ce choix permet d'utiliser l'environnement de liaison et les mécanismes disponibles en Prolog séquentiel pour réaliser l'exécution d'un programme GHC sur une machine séquentielle.

L'utilisation du Prolog séquentiel permet d'avoir une primitive d'unification en GHC assez simple parce que basée sur l'unification Prolog. Le langage Prolog séquentiel offre aussi le moyen de faire des rétroparcours lors du choix de la clause à utiliser pour réduire un objectif. Ce mécanisme peut donc être utilisé tel quel. De plus, certains prédicats prédéfinis en GHC correspondent aux mêmes prédicats prédéfinis en Prolog séquentiel. Il ne faut donc pas les redéfinir.

II.4. Spécification de l'interpréteur

En entrée :

- un système à résoudre sous la forme d'une conjonction d'objectifs (un objectif étant un prédicat GHC),
- un programme GHC,
- un ensemble de prédicats prédéfinis en Prolog séquentiel.

En sortie :

- réussite et effets de bord de la résolution de chacun des objectifs du système en entrée à l'aide du programme GHC et des prédicats prédéfinis.
  Le système en entrée est alors réduit au système vide.

ou

- échec et effets de bord de la résolution des objectifs du système en entrée qui ont été résolus à l'aide du programme GHC et des prédicats prédéfinis.
  Le système en entrée n'est pas réduit au système vide car au moins un des objectifs du système d'entrée ne peut être résolu.

Signalons qu'aucune analyse syntaxique du programme GHC n'est faite. Nous verrons à la section II.5.7. que cette absence d'analyse syntaxique peut engendrer un problème important lors de l'interprétation d'un programme GHC.
Aucune génération de code Prolog simulant le programme GHC puis exécution de ce code Prolog à l'aide de l'interpréteur Prolog n'est effectuée.

II.5. Construction de l'interpréteur

II.5.1. Procédé de construction

Le procédé choisi pour la construction de l'algorithme de l'interpréteur est une conception descendante. A partir des spécifications de l'interpréteur données à la section II.4., des sous-problèmes vont être définis et spécifiés. Le même procédé sera appliqué de façon récursive à chaque sous-problème ainsi identifié, jusqu'au moment où les sous-problèmes pourront être considérés comme étant assez simples que pour être directement codés en Prolog séquentiel.

II.5.2. Premier niveau de raffinement

Il est important de préciser des notions qui seront utilisées tout au long de la description de l'interpréteur : les notions de processus GHC, de types d'objectifs et d'interprétation d'un système.

Un *processus GHC* est un processus qui correspond à l'exécution de la résolution d'un appel (ou objectif) donné, cet appel pouvant apparaître indifféremment dans une garde ou dans un corps de clause GHC.
De par les restrictions apportées au GHC implémenté par l'interpréteur proposé ici (voir section II.2.), le seul cas où une suspension est simulée est celui où on exporte une liaison à l'appel lors de son unification avec une tête de clause. Ce cas est

indépendant du fait que l'appel apparaisse dans la garde ou dans le corps d'une clause.

Si aucune restriction n'avait été apportée au GHC "pur", il aurait fallu distinguer un appel issu d'une garde d'un appel issu d'un corps. En effet, dans ce cas, l'appel issu d'une garde ne peut effectuer (de façon directe ou indirecte) de liaison aux variables de la tête de clause avant que la clause ne soit candidate. Tandis que l'appel issu d'un corps ne peut effectuer de liaison aux variables apparaissant dans la tête de clause ou dans les prédicats de la garde tant que la clause n'est pas candidate. Les conditions de suspension auraient dès lors été différentes.

Exemple : appel : **a(X).**

clause GHC : **a(Y) :- true, !, b(Y), c(Y).**

Un processus GHC est créé pour la résolution de **a(X)**. Ce processus GHC va unifier l'appel et la tête de la clause et ensuite se réduire à deux nouveaux processus GHC, respectivement pour la résolution du sous-objectif **b(Y)** et pour celle du sous-objectif **c(Y)**.

On parlera dans la suite des "processus GHC d'un système", plutôt que des "processus correspondant à la résolution des objectifs d'un système", ceci dans un but évident de simplification de terminologie.

Tous les processus GHC d'une conjonction ne sont pas du même type et ne doivent pas être traités de la même façon. Ils sont regroupés en deux catégories : les prédicats GHC et les prédicats prédéfinis.

Les *prédicats GHC* correspondent à des procédures en GHC. L'ensemble de ces prédicats forme un programme GHC. Les clauses de ces procédures vérifient la syntaxe de GHC telle qu'elle a été définie à la section II.2.6. La résolution d'un objectif correspondant à un prédicat GHC s'effectue de façon à respecter la sémantique procédurale de GHC (cfr section I.5.2.).

Un objectif dans la file d'ordonnancement qui est un prédicat GHC sera, s'il peut être résolu, retiré de la file et réduit au corps d'une clause candidate de la procédure à laquelle il correspond dans le programme GHC.

Exemple : la procédure **reserve** dans l'exemple de réservation aérienne donné à la section I.5.3.

```
reserve(Flight,Seats,DB,Response,DB1) :- true, !,
        value(DB,Flight,Freeseats),
        sub(Freeseats,Seats,Leftseats),
        respond(DB,Leftseats,Flight,Response,DB1).
```

Les *prédicats prédéfinis* correspondent aux prédicats "built-in" par rapport à GHC. Entrent dans cette catégorie les prédicats prédéfinis disponibles pour le Prolog séquentiel et les prédicats ajoutés sous forme de procédures en Prolog séquentiel.

Il est nécessaire de distinguer cette catégorie de prédicats de ceux correspondant à des procédures en GHC car les prédicats prédéfinis ne doivent pas être résolus de la même façon. Ils sont résolus directement grâce au fait que l'environnement de l'interpréteur est le Prolog séquentiel. Il suffit dès lors de faire un appel au méta-prédicat "call" de Prolog afin de résoudre un tel prédicat.

Un objectif correspondant à un tel prédicat dans la file d'ordonnancement des processus restants à résoudre sera, s'il peut être résolu, retiré de la file sans être réduit. On peut donc assimiler ces prédicats à des faits, par opposition à un programme GHC qui correspond à un ensemble de règles.

Exemple : La procédure qui implémente, dans la réservation aérienne, le test des valeurs de ses deux arguments. C'est une procédure écrite en Prolog et contenant un "cut" (à ne pas confondre avec l'opérateur de "trust" de GHC).

lt(X,Y) :- novar(X), novar(Y), !, X < Y.

Voyons maintenant comment peut être définie l'interprétation d'un système GHC.

L'*interprétation d'un système GHC* **S** à l'aide d'un programme **P** peut être décrite de la façon suivante (cfr [Sh83]).

Chaque processus correspondant à un prédicat GHC **A** du système **S** essaie de se réduire à d'autres processus, en utilisant les clauses de **P**. Un processus **A** peut se réduire en trouvant une clause **A1 :- G, !, B** dont la tête **A1** s'unifie avec **A** et dont la garde **G** est satisfaite.

Chaque processus correspondant à un prédicat prédéfini essaie d'être directement résolu.

Le système **S** est résolu quand il est vide, l'interprétation se termine alors avec succès.

Si au moins un des processus ne peut être réduit (GHC) ou résolu (prédéfini), l'interprétation se termine sur un échec car le système **S** ne peut être résolu.

De cette description, on peut déduire la nécessité de disposer pour l'interpréteur de deux processus spéciaux : un processus-ET (qui résoud un système d'objectifs et appelle un processus-OU pour chaque objectif correspondant à un prédicat GHC) et un processus-OU (qui réduit un objectif correspondant à un prédicat GHC). Ces processus correspondent aux deux sous-problèmes résultant du premier niveau de raffinement. Le rôle qu'ils jouent dans la résolution d'un système va être décrit. Les deux processus seront ensuite spécifiés avant d'être raffinés à leur tour.

Au départ de la résolution d'un système **S**, un processus-ET associé à la conjonction d'objectifs **S** est invoqué. Ensuite, la résolution se déroule de la façon suivante :

• un *processus-ET*, invoqué avec un système S, essaie de résoudre directement chaque prédicat prédéfini et crée un processus-OU pour chaque prédicat GHC apparaissant comme objectif de S.

Chaque processus-OU qui se termine avec succès transmet au processus-ET le système B du corps de la clause à utiliser pour la réduction. Celui-ci réduit alors le processus-OU qui s'est terminé en remplaçant par le système B le processus GHC pour lequel le processus-OU avait été créé. De nouveaux processus-OU sont créés pour chacun des processus GHC correspondant à des prédicats GHC dans le système B.

Si un processus-OU se termine sur un échec, il le signale au processus-ET.

Si une résolution de prédicat prédéfini se termine avec succès, le processus GHC correspondant au prédicat prédéfini est retiré du système S.

Quand le processus-ET initial n'a plus de processus GHC dans le système qu'il maintient, il se termine. Il se termine en signalant son propre succès si toutes les résolutions de prédicats prédéfinis et tous les processus-OU créés par lui se sont terminés avec succès. Il se termine sur un échec si au moins une résolution ou un processus-OU s'est terminé sur un échec.

• un *processus-OU*, invoqué avec un prédicat GHC A, procède de la façon suivante. Tant qu'une clause candidate n'est pas trouvée, et qu'il reste des clauses qui n'ont pas encore été évaluées, il essaie d'unifier la tête de la clause suivante dans la procédure correspondant au processus GHC A, soit la clause A1 :- G, !, B.

◇ Si l'unification aboutit et que le processus-ET qu'il a invoqué avec le système G lui a signalé son succès, il signale le succès à son parent-ET, lui transmet le système B et se termine.

◇ Sinon (on se trouve alors dans le cas d'une suspension ou d'un échec), il essaie d'unifier la tête de la clause suivante. S'il n'est pas possible de trouver une clause candidate pour le prédicat GHC A, il signale l'échec à son parent-ET et se termine.

Une spécification plus précise des processus-ET et -OU va maintenant être proposée.

1) Spécification du processus-ET

En entrée :

- un système S de processus GHC à résoudre
- un programme GHC
- des prédicats prédéfinis

En sortie :

- réussite et effets de bord des résolutions de tous les processus du système S d'entrée sur base du programme GHC et des prédicats prédéfinis. Le système S est réduit au système vide.

ou

- échec et effets de bord des résolutions des processus pouvant être résolus en fonction du programme GHC et des prédicats prédéfinis. Le système S n'est pas vide car au moins un des processus de ce système ne peut être résolu.

2) Algorithme réalisant cette spécification du processus-ET

(1) Tant qu'il reste des processus GHC non résolus
choisir un processus GHC non résolu

(1.1) si c'est un prédicat prédéfini
alors essayer de le résoudre directement
(1.1.1) si la résolution réussit
alors on se retrouve dans la situation (1) avec le processus GHC en moins à résoudre
(1.1.2) sinon c'est l'échec de la résolution du système S

(1.2) sinon créer un processus-OU
(1.2.1) si le processus-OU réussit
alors réduire le processus GHC au système du corps communiqué par le processus-OU, on se retrouve dans la situation (1) avec le processus GHC réduit
(1.2.2) sinon, c'est l'échec de la résolution du système S.

3) Spécification du processus-OU

En entrée :

- un processus GHC correspondant à un prédicat GHC à réduire
- un programme GHC
- des prédicats prédéfinis

En sortie :

- réussite et résultat = système du corps de la clause auquel peut être réduit le processus GHC dur base du programme GHC et des prédicats prédéfinis.

ou

- échec et aucun résultat de réduction.

4) Algorithme réalisant cette spécification du processus-OU

(1) Chercher une clause dont la tête s'unifie avec le processus GHC donné en entrée

(1.1) s'il existe une telle clause
alors - créer un processus-ET auquel est communiqué comme système à résoudre la garde de cette clause
- attendre la fin de ce processus-ET

(1.1.1) s'il se termine sur une réussite,
alors le processus-OU se termine sur une réussite et le résultat est le système du corps de cette clause
(1.1.2) sinon on se retrouve dans la situation (1) avec la clause suivante comme nouvelle clause

(1.2) sinon le processus-OU se termine sur un échec.

II.5.3. Raffinement du processus-ET

Un processus-ET est à tout moment responsable d'un ensemble de processus GHC restants à résoudre. En effet, comme chaque réduction de processus GHC correspondant à un prédicat GHC revient à remplacer le processus réduit par un nouveau système de processus GHC à résoudre, et que chaque résolution d'un prédicat prédéfini qui aboutit revient à retirer du système le processus GHC correspondant, le système restant à résoudre est constamment modifié. Ces processus GHC restant à résoudre sont ordonnancés dans une file. Une stratégie d'ordonnancement doit donc être définie pour déterminer l'ordre dans lequel les sous-objectifs seront résolus.

Si la résolution des sous-objectifs d'un système se faisait comme en Prolog séquentiel, ils seraient résolus séquentiellement en partant de la gauche vers la droite, cela ne serait pas du tout non-déterministe-ET par rapport à l'ordre que les sous-objectifs ont dans le système. Il faudrait donc que l'ordre dans lequel les

sous-objectifs d'une conjonction sont résolus n'ait rien à voir avec leur place dans la conjonction.

a. Stratégie d'ordonnancement des objectifs non encore résolus

Le problème de l'ordonnancement des objectifs non encore résolus est semblable au concept d'ordonnancement de la file des noeuds restants à explorer dans les algorithmes généraux de recherche, et plus particulièrement dans les recherches dans des arbres ET/OU (cfr [Ni71]).

Une stratégie d'ordonnancement en largeur d'abord permettrait d'approcher le non-déterminisme-ET car l'ordre dans lequel les sous-objectifs d'une conjonction seraient résolus serait non-déterminé par rapport à la place qu'ils occupent dans la conjonction. L'intérêt de cette stratégie sera mis en valeur grâce à une comparaison avec la stratégie en profondeur d'abord qui est employée par le Prolog séquentiel.

• Voyons la résolution d'une conjonction de sous-objectifs au moyen de la stratégie en profondeur d'abord. Considérons le programme suivant :

```
a :- a1.    (1)
b :- b1.    (2)
c.          (3)
a1 :- a2.
b1.                 conjonction de sous-objectifs :
a2.                              a,b,c.
```

La première étape de la résolution sera de réduire le premier sous-objectif de la conjonction **a** au corps de la clause (1). Le corps **a1** sera placé en tête de conjonction, après avoir retiré le sous-objectif **a**. La conjonction à résoudre devient : **a1,b,c**.

La réduction du premier sous-objectif de la conjonction donne la nouvelle conjonction : **a2,b,c**.

a2 ayant un corps vide, sa réduction correspond à son retrait de la conjonction. Le sous-objectif **a** est donc résolu, la conjonction devient : **b,c**.

La résolution de **b** se déroule de la même façon : on obtient successivement les conjonctions **b1,c** puis **c**.

Le sous-objectif **b** est donc le deuxième à être résolu.

Le sous-objectif **c** est immédiatement résolu et est retiré de la conjonction. La résolution de la conjonction est terminée car elle est ainsi réduite à la clause vide.

L'ordre de résolution des différents sous-objectifs correspond donc à l'ordre dans lequel ils apparaissent dans la conjonction initiale : **a,b,c**.

Il faut remarquer que si le sous-objectif **a** était réductible à l'infini, on n'essaierait même pas les autres sous-objectifs car ils n'atteindraient jamais la tête de la conjonction. Cette stratégie n'est donc pas équitable-ET.

• Le même problème va servir de base à l'illustration de la résolution d'une conjonction suivant la stratégie en largeur d'abord.

La réduction du premier sous-objectif **a** donne comme résultat **a1**. Ce nouveau sous-objectif est ajouté en fin de conjonction et le premier sous-objectif **a** est retiré. Cela donne la conjonction : **b,c,a1**.

Le premier sous-objectif de la conjonction est réduit selon la même stratégie, la conjonction résultante est **c,a1,b1**.

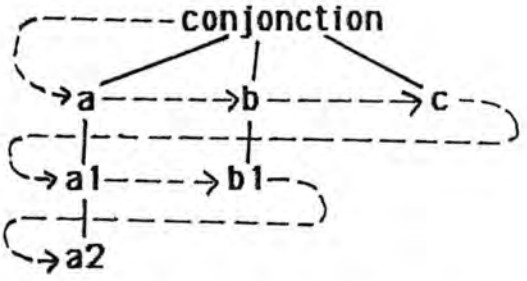
Le sous-objectif **c** est immédiatement résolu et est retiré de la conjonction qui devient **a1,b1**.

a1 est réduit et le résultat est **b1,a2**.

b1 est immédiatement résolu, la résolution du sous-objectif **b** est donc terminée.

a2 est ensuite résolu, le sous-objectif **a** clôture ainsi la résolution de la conjonction.

L'ordre de résolution des différents sous-objectifs ne correspond pas à l'ordre dans lequel ils sont situés dans la conjonction initiale **(a,b,c) ≠ (c,b,a)**. En fait, le premier sous-objectif à être résolu est celui dont la descendance est la moins profonde dans l'arbre de résolution ET/OU. Soit l'arbre ET/OU du programme de l'exemple :



On voit clairement que la résolution de **c** est immédiate car elle ne nécessite pas de descendre d'un niveau. Celle de **b** nécessite la descente d'un niveau, et celle de **a**, de deux niveaux.

• La stratégie d'ordonnancement en largeur d'abord permet en quelque sorte d'approcher un "non-déterminisme-ET" car l'ordre dans lequel les sous-objectifs d'une conjonction sont résolus est non-déterminé par rapport à l'ordre qu'ils ont dans la conjonction. L'ordre dans lequel ils sont résolus étant cependant prévisible, l'utilisation de cette stratégie n'est qu'une petite approche du non-déterminisme-ET car pour chaque résolution de la conjonction, l'ordre dans lequel les sous-objectifs sont résolus sera le même. Cette stratégie est équitable.

La place mémoire nécessaire pour la stratégie en largeur d'abord est plus grande car la conjonction comporte à certains moments beaucoup plus d'objectifs qu'en ce qui concerne la stratégie en profondeur d'abord. On peut aussi signaler que la stratégie d'ordonnancement en largeur d'abord est peut-être moins efficace que celle en profondeur d'abord dans certains cas car des réductions de sous-objectifs risquent d'être effectuées inutilement.

Un compromis existe entre ces deux stratégies : l'ordonnancement en profondeur d'abord limité à N niveaux ("N-bounded depth-first scheduling"), mais l'utilisation de cette stratégie aurait considérablement complexifié l'ordonnancement sans pour autant proposer une meilleure approche du non-déterminisme-ET. Elle n'a donc pas été retenue ici.

La stratégie d'ordonnancement utilisée pour l'interpréteur de GHC est l'ordonnancement en largeur d'abord. La principale caractéristique ayant guidé ce choix est la possibilité d'approcher (même si c'est de loin) le non-déterminisme-ET. Shapiro [Sh83] a également basé son interpréteur de Concurrent Prolog sur la stratégie en largeur d'abord.

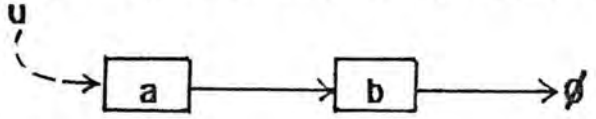
Une modification de cette stratégie d'ordonnancement offrant un vrai non-déterminisme-ET sera proposée à la section II.7.

Un processus-ET doit donc gérer une file d'ordonnancement contenant les processus GHC non encore résolus. Cette file est implémentée sous forme d'une liste de différence de listes. Une présentation de cette représentation de données est effectuée au point suivant.

b) Représentation de la file d'ordonnancement : liste de différence de listes

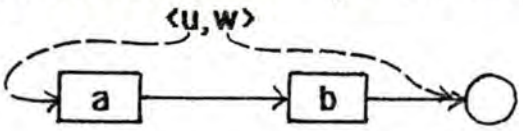
Ce concept de représentation d'une liste par une différence de listes est introduit dans [ClTa77] en se basant sur la représentation interne d'une liste.

Une liste est représentée de façon interne par un pointeur vers une chaîne d'éléments, comme dans la figure suivante ($\emptyset$ représente la liste vide)



La liste représentée par la valeur du pointeur **u** contient tous les éléments de la chaîne commençant à a, et se terminant avec le pointeur vers la liste vide. La liste représentée est donc [a,b].

On peut aussi représenter une liste par une paire de pointeurs, une illustration de cette technique est présentée à la figure suivante :



où ◯ représente soit une liste d'éléments, soit la liste vide.

La paire de pointeurs ⟨u,w⟩ est une représentation de la liste [a,b]. Cette représentation est basée sur la convention que la liste ⟨u,w⟩ a pour éléments tous les éléments de la chaîne commençant par l'élément pointé par u et se terminant par l'élément précédant directement la structure pointée par w. w peut être un pointeur vers n'importe quelle structure (liste d'éléments ou liste vide).

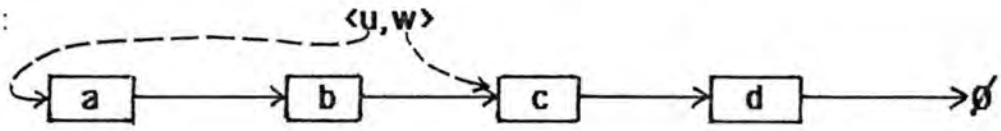
Une liste de différence de listes (liste-d, pour abréger) peut être définie de la façon suivante :

"toute paire ⟨u,w⟩ est une liste-d ssi u = w (auquel cas elle est vide)
ou u = x . v
où x est un élément de liste
et ⟨v,w⟩ une liste-d."

Dans l'expression 'x . v', l'opérateur '.' est le constructeur de liste qui prend un élément (x) et une liste (v) et construit une nouvelle liste (x.v).

Dans le cadre de l'interpréteur, la paire de pointeurs d'une liste-d est représentée par la paire de listes représentées séparément par chacun des deux pointeurs.

Exemple :



La liste représentée par ⟨u,w⟩ sera définie par la paire de listes (tête,queue) :

tête = [a,b,c,d]
queue = [c,d].

Les éléments de la liste représentée par ⟨u,w⟩ sont les éléments qui appartiennent à la liste tête et n'appartiennent pas à la liste queue. La liste considérée est donc le résultat de la "différence" entre les deux listes tête et queue.

L'intérêt de cette représentation de liste est la possibilité d'ajouter un élément en fin de liste et de concaténer deux listes sans avoir de recours explicite à l'opérateur de concaténation de listes.

Exemples :

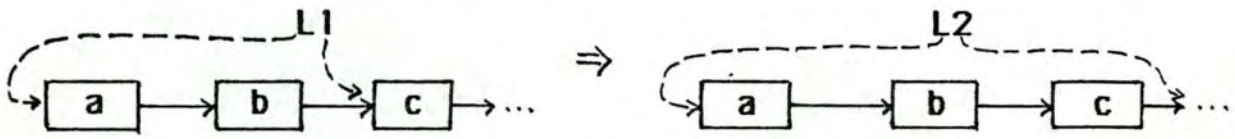
1) ajout d'un élément en fin de liste

1.1.) liste traditionnelle (représentée par un pointeur unique)

liste d'origine : L1 = [a,b]
élément à ajouter : c
liste finale : L2 = liste résultant de l'opération de concaténation
append(L1,[c],L2)
= [a,b,c]

1.2.) liste-d

liste d'origine : L1 = { [a,b,c|Q] , [c|Q] } où Q est une liste quelconque
= [a,b]
élément à ajouter : c
liste finale : L2 = { [a,b,c|Q] , Q } où Q reste inchangée
= [a,b,c]



2) concaténation de deux listes

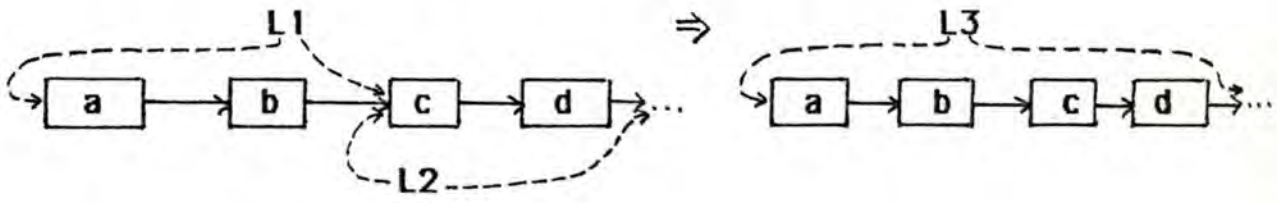
2.1.) listes traditionnelles

listes d'origine : L1 = [a,b]
L2 = [c,d]
liste finale : L3 = liste résultat de l'opération de concaténation
append(L1,L2,L3)
= [a,b,c,d]

2.2.) listes-d

listes d'origine : L1 = { [a,b,c,d|Q] , [c,d|Q] }
L2 = { [c,d|Q], Q } où Q est une liste quelconque

liste finale : L3 = [[a,b,c,d|Q], Q] où Q est inchangée



Il était important de trouver une manipulation de listes qui ne soit pas trop lourde car la liste d'ordonnancement est très fréquemment modifiée (de par la stratégie en largeur d'abord) pour ajouter un élément en fin de liste ou pour enlever un élément en tête de liste. Toutes les manipulations de cette liste peuvent donc être effectuées sans faire de référence à la primitive de concaténation. Cette même représentation de listes est utilisée dans l'interpréteur construit par Shapiro dans [Sh83].

Remarque : enlever un élément en tête de liste ne nécessite aucune primitive explicite, quelle que soit la représentation de la liste. Il suffit de considérer l'ancienne liste comme [a|L], et la nouvelle liste sera tout simplement L. La représentation de liste par différence de listes n'apporte pas de simplification à cette manipulation de liste.

c) Détection de l'échec cyclique d'un processus GHC

Il a été signalé à la section II.2.2. qu'aucune distinction n'est faite entre échec et suspension. Quand un processus-OU créé par un processus-ET lui communique son échec, cela peut signifier un échec réel ou une demande de suspension. Le processus GHC correspondant est alors reporté à la fin de la file d'ordonnancement, car un processus GHC suspendu est un processus non encore résolu. L'échec de la résolution d'un prédicat prédéfini est aussi traité de la même façon : le processus GHC correspondant est reporté en fin de file d'ordonnancement.

Le traitement des suspensions et des échecs de façon semblable peut entraîner des problèmes.

En effet, un report en fin de file d'ordonnancement pour traiter un échec ne changera rien à la situation d'échec. Le processus GHC ayant définitivement échoué, le résultat de la résolution des autres processus non encore résolus n'apportera aucune modification permettant au processus ayant échoué de réussir lors d'un prochain essai de résolution. Ce processus va donc rester indéfiniment dans la file en étant chaque fois reporté à la fin, puisqu'il ne peut être résolu.

De par l'absence de distinction entre échec et suspension, cette situation d'échec d'un processus est semblable à celle de blocage individuel permanent d'un processus

dû à une suspension (cfr section I.1.8.). En effet, si un processus P1 est suspendu en attendant une valeur pour une variable partagée et si aucun processus producteur n'existe pour cette variable, P1 restera suspendu indéfiniment. En effet, P1 sera reporté en fin de file à chaque essai de résolution.

Ces deux cas (échec et suspension infinie) posent le même problème : il y a un processus GHC dans la file d'ordonnancement qui ne peut être résolu. Ce processus restant indéfiniment dans la file d'ordonnancement, celle-ci ne sera jamais vide et la résolution de l'appel initial ne se terminera pas. Cette situation d'échec cyclique doit être détectée afin que l'erreur soit signalée et que la résolution de l'appel prenne fin. Cette détection est réalisée grâce au mécanisme décrit ci-dessous.

Mécanisme de détection d'échec cyclique

1) Rôle du mécanisme

Le mécanisme de détection d'échec cyclique a pour rôle de détecter une éventuelle situation d'échec cyclique afin de pouvoir alors faire échouer la résolution de l'appel initial.

Une situation d'échec cyclique se présente lorsque tous les processus non encore résolus restants dans la file d'ordonnancement ont déjà été sélectionnés au moins une fois pour être résolus. Aucun de ces essais de résolution n'a abouti (que ce soit à cause d'un échec ou d'une suspension). Ils ont donc tous été replacés dans la file d'ordonnancement. Si aucun de ces processus ne peut être résolu et que la cause en est un échec ou une suspension infinie, nous nous trouvons devant une situation d'échec cyclique car la file d'ordonnancement ne se videra jamais : pour tout essai de résolution, chaque processus sera reporté en fin de file.

Ce mécanisme n'est donc pas destiné à éviter de réévaluer plusieurs fois des processus voués à l'échec, mais bien à éviter de poursuivre toute évaluation lorsque tous les processus de la file d'ordonnancement sont voués à l'échec. Un processus ayant échoué pourra dès lors être évalué plusieurs fois.

2) Principe du mécanisme

Le principe du mécanisme de détection d'échecs cycliques repose sur la différenciation des trois situations suivantes : situation initiale d'un cycle, situation générale en cours de cycle, et situation finale d'un cycle.

La *situation initiale* est celle où aucun des processus rangés dans la file d'ordonnancement n'a encore été résolu.

La *situation générale* en cours de cycle est celle où certains processus ont déjà été résolus et où d'autres ont peut-être déjà été reportés dans la file pour

cause d'échec ou de suspension. Cependant, tous les processus de la file initiale n'ont pas encore été choisis pour essayer d'être résolus.

Même si tous les processus pour lesquels un essai de résolution a déjà été effectué ont été reportés dans la file, le plus important est qu'il reste des objectifs qui n'ont pas encore été évalués. En effet, si au moins un des objectifs a été reporté en fin de file pour raison de suspension, la résolution d'un des processus non encore évalués permettra peut-être de supprimer cette suspension (par assignation par un producteur d'une valeur à une variable partagée). Dans ce cas, une prochaine évaluation du processus suspendu pourrait se terminer avec succès. On ne se trouve donc pas dans une situation d'échec cyclique.

La *situation finale* d'un cycle est celle où tous les processus ont été évalués.

Si les processus se trouvant dans la file d'ordonnancement sont exactement les mêmes que ceux qui s'y trouvaient lors de la situation initiale du cycle, cela signifie qu'aucun d'eux ne peut être résolu. Nous nous trouvons dès lors devant une situation d'échec cyclique.

Si par contre la file d'ordonnancement de la situation finale n'est pas la même que celle de la situation initiale, cela signifie qu'au moins un processus a été résolu et retiré de la file. La résolution de ce processus pouvant supprimer une suspension pour un des processus restants dans la file, il faut considérer la file telle qu'elle est comme la file d'une situation initiale d'un cycle. Chaque processus de cette file devra être (ré)évalué.

3) Implémentation du mécanisme

Situation initiale d'un cycle

Les objectifs correspondant au système à résoudre sont ordonnancés dans une liste vide et on ajoute à la fin de celle-ci l'élément artificiel 'cycle' qui servira de marque pour identifier la situation finale d'un cycle.

La variable E (indicateur d'échec cyclique) sera utilisée pour pouvoir vérifier si un changement est intervenu entre la file de la situation initiale et celle de la situation finale. On lui assigne au début la valeur 'échec' et dès qu'un objectif est résolu, cette valeur devient 'pas_échec'.

Exemple : la conjonction des deux objectifs B et C doit être résolue

file d'ordonnancement :

B	C	cycle

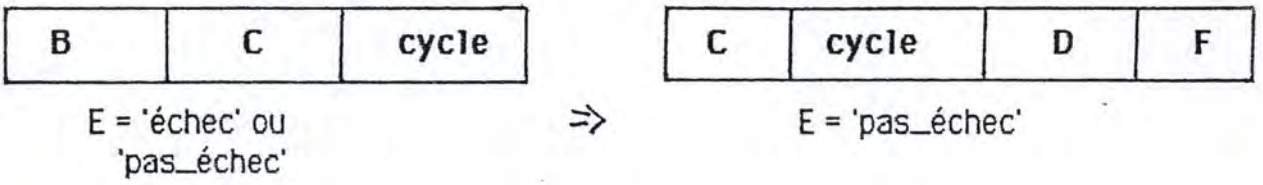
variable E : 'échec'

Situation générale en cours de cycle

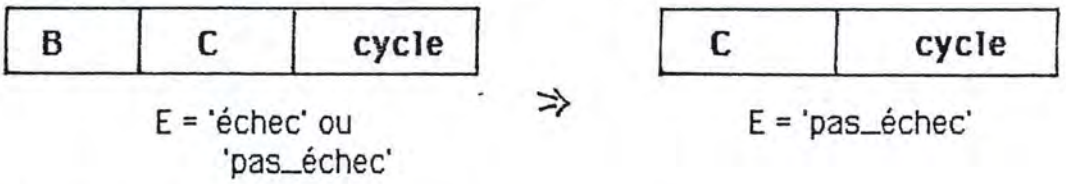
Premier cas : réussite de l'évaluation d'un objectif de la file

Le premier processus B de la liste réussit (c'est-à-dire est résolu directement si c'est un prédicat prédéfini ou est réduit à la conjonction de sous-objectifs D,F si c'est un prédicat GHC). La valeur de la variable E est modifiée en 'pas_échec' car il y a eu un changement dans la file d'ordonnancement susceptible d' apporter des éléments nouveaux pour la résolution des autres objectifs (une variable peut avoir été instanciée si l'objectif est résolu, ou le sera peut-être via les nouveaux sous-objectifs ordonnancés si l'objectif considéré est réduit).

Exemple 1 : B est un prédicat qui peut être réduit à la conjonction D,F. En fonction de la stratégie en largeur d'abord, les deux nouveaux objectifs à résoudre sont placés en fin de file d'ordonnancement.



Exemple 2 : B est un prédicat prédéfini qui peut être résolu. Il est exécuté directement et est retiré de la file.



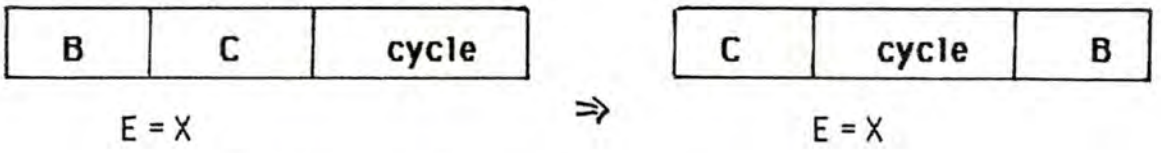
Second cas : échec de l'évaluation d'un objectif

Deux raisons peuvent être à l'origine de cet échec de l'évaluation.

Le processus B en tête de liste voudrait exporter une liaison qui enfreindrait les règles de suspension. L'unification échoue et le processus est suspendu en étant reporté en fin de liste.

Le processus B en tête de liste échoue définitivement. Par manque de distinction entre échec et suspension, il est aussi reporté en fin de file d'ordonnancement.

Dans les deux cas, la variable E n'est pas modifiée car la file n'a subi aucun changement susceptible de supprimer une situation d'échec (aucun objectif n'a été réduit, ni résolu).



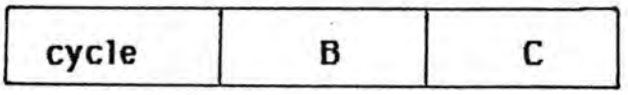
E = X ⇒ E = X

où X = 'échec' ou 'pas_échec'

Situation finale d'un cycle

Premier cas : détection d'échec cyclique

L'élément 'cycle' se retrouve en tête de file et la valeur de la variable E est toujours 'échec'. Cela signifie qu'il n'y a eu aucune réduction ou résolution d'objectif dans la file, bien que tous les objectifs aient été évalués. Il n'y a dès lors plus aucune chance de résoudre les objectifs restants. Cette détection d'échec cyclique implique l'arrêt immédiat et l'échec de la résolution du système de l'appel concerné.



E = 'échec'

Second cas : pas d'échec cyclique

L'élément 'cycle' se retrouve en tête de file et la valeur de la variable E est 'pas_échec', il n'y a donc aucun problème car il y a eu une résolution ou réduction d'objectif. L'élément 'cycle' est alors reporté en fin de file et la valeur de la variable E est modifiée en 'échec', de façon à se trouver dans la situation initiale d'un cycle pour commencer la résolution de la file modifiée.

E = 'pas_échec' ⇒ E = 'échec'

d) Mise en évidence des sous-problèmes du processus-ET

Un processus-ET correspondra à la procédure solve(Goals) de l'interpréteur.

L'argument Goals est le système à résoudre, représenté sous la forme d'une liste d'objectifs.

De façon à déterminer l'ordre dans lequel les objectifs doivent être considérés pour être résolus, il est nécessaire de les placer dans une file d'ordonnancement en

appliquant la stratégie d'ordonnancement adoptée ici : une stratégie en largeur d'abord. La résolution considèrera ces objectifs en fonction de la place qu'ils occupent dans la file d'ordonnancement, et veillera à détecter une éventuelle situation d'échec cyclique (voir point c de cette section).

Deux sous-problèmes peuvent être dégagés de cette description d'un processus-ET :

1) Ordonnancer les sous-objectifs d'une conjonction dans une file d'ordonnancement donnée. On appellera ce sous-problème 'schedule'.

Le sous-problème schedule correspond à la procédure schedule(A,Head,Tail,Newhead,Newtail) de l'interpréteur. Le premier argument est le système qui doit être ordonnancé, Head et Tail représentent (par liste de différence de listes) la file avant l'ordonnancement, Newhead et Newtail représentent la même file après ordonnancement.

2) Résoudre les processus GHC d'une file d'ordonnancement donnée, en détectant une éventuelle situation d'échec cyclique. Ce sous-problème s'appellera 'ghc_solve'.

Le sous-problème ghc_solve correspondra à la procédure ghc_solve(Head,Tail,E) de l'interpréteur. Les deux premiers arguments (Head + Tail) représentent la file d'ordonnancement contenant le système à résoudre, sous forme d'une liste de différence de listes. Le troisième argument E est la variable servant d'indicateur d'échec cyclique.

Ces deux sous-problèmes peuvent être spécifiés plus précisément :

1) spécification de schedule(A,Head,Tail,Newhead,Newtail)

La procédure schedule(A,Head,Tail,Newhead,Newtail) établit la relation : "Newhead, Newtail est la liste de différence de listes correspondant à la file d'ordonnancement Head,Tail dans laquelle ont été ordonnancés les sous-objectifs du système A, conformément à une stratégie d'ordonnancement en largeur d'abord."

2) spécification de ghc_solve(Head,Tail,E)

<u>En entrée</u> :
- une file d'ordonnancement représentée sous la forme d'une liste de différence de listes et contenant un système de processus GHC à résoudre (Head, Tail).
- une variable E instanciée à la valeur 'échec' ou 'pas_échec'.
- le programme GHC et les prédicats prédéfinis.

En sortie :
- réussite et file d'ordonnancement vide, variable E instanciée à 'pas_échec', et effets de bord des résolutions des processus GHC de la file d'entrée en fonction du programme GHC et des prédicats prédéfinis.

ou

- échec et file d'ordonnancement non vide, variable E instanciée à 'échec', et effets de bord des résolutions des processus GHC ne se trouvant plus dans la file (en fonction du programme GHC et des prédicats prédéfinis).

3) algorithme réalisant cette spécification de ghc_solve

(1) Tant qu'il reste au moins un objectif dans la file

(1.1) si le(s) objectif(s) restant(s) dans la file ne détermine(nt) pas une situation d'échec cyclique
alors

(1.1.1) si le premier processus est un prédicat prédéfini
alors essayer de le résoudre
(1.1.1.1) si la résolution aboutit
alors - le retirer de la file
- assigner à E la valeur 'pas_échec'
(1.1.1.2) sinon le reporter à la fin de la file

(1.1.2) sinon (prédicat GHC)
essayer de le réduire
(1.1.2.1) si la réduction réussit
alors - le retirer de la file
- insérer le système du corps de la clause auquel il peut être réduit en bout de file d'ordonnancement en respectant la stratégie d'ordonnancement en largeur d'abord
- assigner à E la valeur 'pas_échec'
(1.1.2.2) sinon, le reporter à la fin de la file

(1.2) sinon (échec cyclique)
faire échouer la résolution.

On obtient dès lors le fragment de code suivant pour la procédure solve(Goals) :

```
solve(Goals) :- schedule(Goals,X,X,Head,[cycle|Tail]),
                ghc_solve(Head,Tail,échec).
```

Ces deux sous-problèmes (schedule et ghc_solve) vont être raffinés, avant de passer au raffinement du processus-OU. La continuité du raisonnement suivi apparaîtra ainsi plus clairement.

II.5.4. Construction de la procédure schedule(A,Head,Tail,Newhead,Newtail)

Une stratégie d'ordonnancement en largeur d'abord consiste à ajouter les sous-objectifs du système un à un à la fin de la file d'ordonnancement (cfr point a de la section II.5.3.).

La *situation générale* est celle où il reste des objectifs à ordonnancer. Ces objectifs sont considérés un à un de la façon suivante : on fait tout d'abord l'ordonnancement du premier élément de la liste en le plaçant en fin de liste grâce à l'utilisation de liste de différence de listes. La liste résultant de ce premier ordonnancement sera utilisée comme file d'ordonnancement en entrée pour ordonnancer de façon récursive la queue de la liste d'objectifs.

La *condition d'arrêt* est celle où tous les objectifs de la conjonction ont été ordonnancés dans la file. La liste A étant vide, il n'y a plus rien à faire.

Les clauses permettant d'implémenter cet ordonnancement sont au nombre de deux.

```
schedule([],Head,Tail,Head,Tail).
schedule([Goal|List],Head,[Goal|Tail],Newhead,Newtail) :-
                schedule(List,Head,Tail,Newhead,Newtail).
```

II.5.5. Construction de la procédure ghc_solve(Head,Tail,E)

La construction de cette procédure sera basée sur les descriptions de la situation générale et de la condition d'arrêt qui seront faites ci-dessous.

La *situation générale* peut être caractérisée par une série de situations particulières correspondant aux différents cas identifiés lors de la présentation du mécanisme de détection d'échecs cycliques (point c, section II.5.3.).

Le premier cas correspond à la situation finale d'un cycle où il n'y a pas d'échec cyclique. Cette situation est caractérisée par l'élément 'cycle' qui se trouve en tête de file d'ordonnancement et par la valeur 'pas_échec' de la variable E.

L'élément 'cycle' doit être reporté en fin de file et la valeur 'échec' doit être assignée à la variable E. Les trois arguments ainsi mis-à-jour seront ceux avec lesquels la résolution sera récursivement invoquée pour résoudre la nouvelle file.

La clause décrivant ce cas est la suivante :

```
ghc_solve([cycle|Head],[cycle|Tail],pas_échec) :-
                        !, ghc_solve(Head,Tail,échec).
```

Le deuxième cas correspond à la situation finale d'un cycle où un échec cyclique a été détecté. Cette situation est caractérisée par l'élément 'cycle' qui se trouve en première position de la file d'ordonnancement et par la valeur 'échec' de la variable E.

Il faut alors faire échouer la résolution. Comme il faut supprimer toute possibilité de rétroparcours qui entraînerait l'essai pour la résolution des autres clauses de la procédure, la combinaison "cut-fail" est nécessaire.

La clause correspondant à ce deuxième cas est la suivante :

```
ghc_solve([cycle|Head],Tail,échec) :- !, fail.
```

Le troisième cas correspond à la situation générale en cours de cycle où le premier objectif de la file est un prédicat prédéfini qui est résolu avec succès.

Il faut alors résoudre récursivement la file d'ordonnancement dont on a retiré le premier élément avec l'argument E instancié à la valeur 'pas_échec'. S'il est impossible de résoudre la nouvelle file, il ne faut en aucun cas faire un rétroparcours sur l'exécution du prédicat prédéfini. C'est pourquoi un "cut" doit précéder l'appel récursif.

La clause décrivant ce cas est la suivante :

```
ghc_solve([G|Head],Tail,E ) :- builtin(G), call(G), !,
                        ghc_solve(Head,Tail,pas_échec).
```

Le quatrième cas correspond à la situation générale en cours de cycle où le premier objectif de la file est un prédicat prédéfini dont la résolution a échoué.

Ce prédicat doit alors être replacé en fin de file et il faut résoudre récursivement cette file mise-à-jour sans modifier la valeur de l'argument E. Un échec éventuel de la résolution de la nouvelle file ne doit pas entraîner l'essai d'une autre clause de la procédure. C'est pourquoi un "cut" précèdera directement l'appel récursif.

La clause décrivant ce cas sera placée dans la procédure juste après la clause décrivant le cas précédent. Ainsi, si la clause précédente échoue, le mécanisme de rétroparcours de Prolog fait que celle-ci sera essayée. L'échec de la clause précédente entraînant un rétroparcours ne peut être dû qu'à un échec de résolution du prédicat qui se trouve au début de la file d'ordonnancement, du moins si celui-ci est de type prédéfini.

Ce cas est décrit par la clause :

```
ghc_solve([G|Head],[G|Tail],E) :- builtin(G), !,
                    ghc_solve(Head,Tail,E).
```

Le cinquième cas correspond à la situation générale en cours de cycle où le premier objectif de la file est un prédicat GHC et peut être réduit par un processus-OU.

Il faut alors ordonnancer les objectifs formant le système du corps auquel il a été réduit dans la file d'ordonnancement dont on a retiré le premier élément. Un appel récursif est ensuite effectué, de façon à résoudre cette nouvelle file. L'argument E est instancié à 'pas_échec' pour cet appel récursif.

La clause suivra directement les deux clauses précédentes dans le corps de la procédure. Ainsi, le mécanisme de rétroparcours de Prolog n'envisagera cette clause-ci que si le premier prédicat de la file d'ordonnancement est un prédicat GHC.

Il est nécessaire de placer un "cut" après l'invocation du processus-OU car, s'il est impossible de résoudre la nouvelle file, il ne faut en aucun cas faire un rétroparcours après la réduction de l'objectif. En effet, la sémantique de GHC spécifie qu'il n'y a aucun rétroparcours sur le choix de la clause candidate. Si un échec apparaît dans la résolution du corps d'une clause, c'est donc aussi l'échec pour l'appel de cette clause, aucune alternative ne doit être envisagée.

La clause décrivant cette situation est la suivante :

```
ghc_solve([G|Head],Tail,E) :-
        reduce(G,R), !,
        schedule(R,Head,Tail,Newhead,Newtail),
        ghc_solve(Newhead,Newtail,pas_échec).
```

Le sixième cas correspond à la situation générale en cours de cycle où le premier objectif de la file est un prédicat GHC qui ne peut être réduit par un processus-OU.

Il faut alors le reporter en fin de file d'ordonnancement et appeller récursivement la résolution avec la variable E inchangée, de façon à résoudre la file d'ordonnancement ainsi modifiée.

Cette clause sera placée juste après la précédente dans le corps de la procédure. Elle ne sera ainsi utilisée (en fonction de l'application du mécanisme de rétroparcours) que si le premier élément de la file n'est pas un processus prédéfini et ne peut être réduit.

Cette situation est décrite par la clause suivante :

```
ghc_solve([G|Head],[G|Tail],E) :- ghc_solve(Head,Tail,E).
```

La *condition d'arrêt* correspond à la situation où la liste d'ordonnancement contenant les objectifs à résoudre ne contient plus que l'élément 'cycle', tous les processus sont donc déjà résolus et il n'y a plus rien à faire. La clause décrivant

cette condition d'arrêt sera la première de la procédure, de façon à être essayée en premier lieu.

La clause suivante décrit la condition d'arrêt :

```
ghc_solve([cycle],[],_) :- !.
```

Le regroupement des différentes clauses décrites pour chaque situation particulière nous permet de proposer le code complet de la procédure ghc_solve(H,T,E).

```
ghc_solve([cycle],[],_) :- !.
ghc_solve([cycle|Head],[cycle|Tail],pas_échec) :- !,
            ghc_solve(Head,Tail,échec).
ghc_solve([cycle|Head],Tail,échec) :- !, fail.
ghc_solve([G|Head],Tail,E) :- builtin(G),
                              call(G), !,
                              ghc_solve(Head,Tail,pas_échec).
ghc_solve([G|Head],[G|Tail],E) :- builtin(G), !,
                                  ghc_solve(Head,Tail,E).
ghc_solve([G|Head],Tail,E) :- reduce(G,R), !,
                      schedule(R,Head,Tail,Newhead,Newtail),
                      ghc_solve(Newhead,Newtail,pas_échec).
ghc_solve([G|Head],[G|Tail],E) :- ghc_solve(Head,Tail,E).
```

II.5.6. Raffinement du processus-OU

Quand un processus-OU est invoqué avec comme paramètre un processus GHC, il doit essayer de trouver une clause candidate pour résoudre ce processus GHC. Cette recherche d'une clause candidate peut être décomposée en deux phases.

1) La première phase est de trouver une clause GHC dont la tête s'unifie avec le processus GHC, ce que l'on dénotera : *guarded_clause*.

Le sous-problème 'guarded_clause' correspondra à la procédure guarded_clause(Goal,Guard,Body).

Goal est l'objectif à résoudre pour lequel il faut trouver la garde d'une clause avec la tête de laquelle il s'unifie. Les deux arguments suivants sont les résultats de la résolution de 'guarded_clause' : Guard est le système de la garde d'une clause dont la tête s'est unifiée à Goal et Body est le système du corps de cette même clause.

Ce sous-problème guarded_clause peut être spécifié de la façon suivante. La procédure guarded_clause(Goal,Guard,Body) établit la relation : " Guard et Body sont deux listes représentant respectivement la conjonction d'objectifs de la garde et

celle du corps de la première clause du programme GHC dont la tête s'unifie avec Goal. "

2) La seconde phase est de résoudre la garde de cette clause en invoquant un processus-ET, ce que l'on dénotera *solve*.

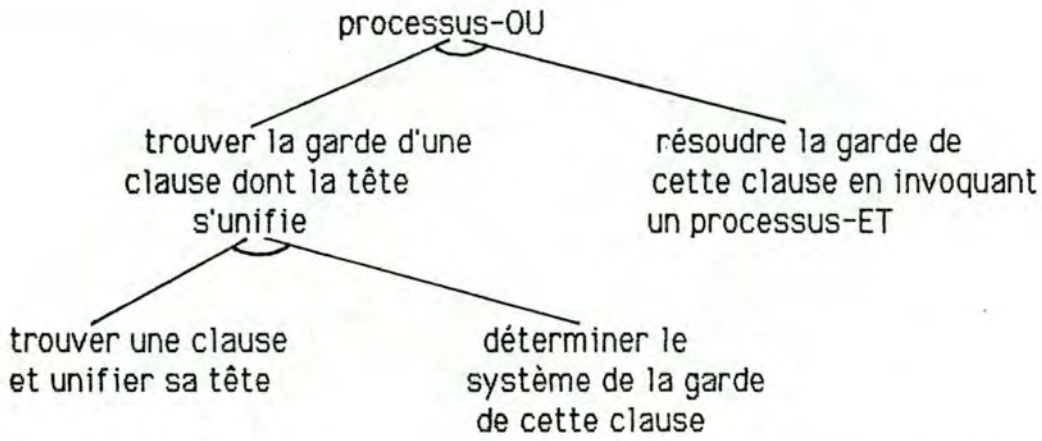
Le sous-problème solve correspond au sous-problème du processus-ET, représenté dans l'interpréteur par la procédure solve(Goals) déjà décrite au point d de la section III.5.3.

Si une clause considérée est non-candidate, on recherche la clause suivante susceptible d'être candidate. S'il n'y a plus de nouvelle clause à évaluer et qu'elles sont toutes non-candidates, le processus-OU échoue. Sinon, il communique au processus-ET qui est son parent le système du corps de la clause qui est candidate.

La recherche effectuée par la résolution du prédicat guarded_clause(Goal,Guard,Body) peut elle-même être décomposée en deux sous-problèmes :
1) trouver une clause dont la tête s'unifie avec le processus GHC, ce que l'on dénotera *find_clause*.
2) déterminer le système de la garde de cette clause, ce que l'on dénotera *find_guard*.

Ces deux sous-problèmes seront décrits lors du raffinement du sous-problème guarded_clause, à la section II.5.7.

La structure d'un processus-OU peut dès lors être schématisée par l'arbre proposé ci-dessous :



Il y a trois raisons différentes pour lesquelles un processus-OU peut échouer :

(a) aucune des clauses n'a de tête qui puisse être unifiée avec le processus GHC sans enfreindre la règle de suspension (a). Il faut donc attendre qu'un autre objectif instancie la(les) variable(s) qui pose(nt) le problème. Ce cas correspond à une suspension. Lorsque le processus-ET a reçu le signal d'échec du processus-OU, il traite la suspension en reportant le processus GHC considéré en fin de file d'ordonnancement.

(b) aucune unification de tête de clause ne réussit car le processus GHC ne correspond à aucune procédure définie dans le programme GHC considéré. Ce cas correspond à un échec et sera traité au niveau du processus-ET comme une suspension. Dès que le processus-OU aura signalé son échec, le processus GHC considéré sera reporté en fin de file d'ordonnancement.

(c) aucune clause dont l'unification de la tête s'est bien passée n'a une garde qu'il soit possible de résoudre. Cette situation est semblable à celle du point (b) ci-dessus : l'échec sera traité par le processus-ET comme une suspension. Dès que le processus-OU aura signalé son échec, le processus GHC considéré sera reporté en fin de file d'ordonnancement.

Un processus-OU correspond à la procédure reduce(Goal,Reduced) de l'interpréteur. Goal représente l'objectif à réduire et Reduced est le résultat de la réduction de Goal : le corps de la clause candidate sélectionnée sous forme d'une liste de sous-objectifs.

Comme on l'a vu, on essaie, via le sous-objectif guarded_clause(Goal,Guard,Reduced), de trouver une clause dont la tête s'unifie avec Goal, puis on résoud la garde correspondante grâce au sous-objectif solve(Guard).

Si le système de la garde ne peut être résolu (échec de la résolution du prédicat solve(Guard)), le mécanisme de rétroparcours de Prolog réévalue le sous-objectif guarded_clause. Cette réévaluation va considérer la clause GHC suivante de la procédure correspondante dans le programme GHC. S'il n'y a plus de nouvelle clause GHC à évaluer, le sous-objectif va échouer à son tour, et ce sera l'échec pour le processus-OU. Dans ce cas, la valeur de la variable Reduced reste indéterminée car cette variable n'est pas instanciée. Cette utilisation du rétroparcours implémenté en Prolog permet d'avoir une recherche séquentielle des clauses GHC d'une procédure.

Si la garde est satisfaite (réussite de la résolution du prédicat solve(Guard)), l'argument Reduced contient la liste des sous-objectifs du corps de la clause sélectionnée. Cette clause sera donc toujours la première (dans l'ordre où le programme GHC a été écrit) pour laquelle l'unification de la tête et l'évaluation de la garde réussissent.

La clause de cette procédure reduce(Goal,Reduced) est la suivante :

reduce(Goal,Reduced) :- guarded_clause(Goal,Guard,Reduced),
solve(Guard).

Des deux sous-problèmes correspondant à la description d'un processus-OU, un seul doit encore être raffiné puisque le sous-problème solve(Guard) correspond au processus-ET déjà défini au point II.5.3. L'étape suivante est donc le raffinement du sous-problème guarded_clause.

II.5.7. Raffinement du sous-problème guarded_clause

Comme cela a été signalé à la section II.5.6., guarded_clause peut à son tour être raffiné en deux sous-problèmes.

Le *sous-problème find_clause* correspondra à la procédure find_clause(Goal,Clause) de l'interpréteur. Goal est l'objectif pour lequel on recherche une clause et Clause est la liste reprenant les sous-objectifs du corps de cette clause.

La spécification de ce sous-problème est la suivante. La procédure find_clause(Goal,Clause) établit la relation : " Clause est la liste dont chaque élément est un sous-objectif du corps (garde GHC + opérateur de "trust" + corps GHC) de la première clause GHC trouvée dont la tête s'unifie avec Goal en respectant la sémantique de GHC. "

Le *sous-problème find_guard* correspondra à la procédure find_guard(Clause,Guard,Body). Le premier argument (Goal) est une liste d'objectifs qu'il faut séparer en deux parties sur base de la place de l'opérateur de "trust" (' ! '). Cet opérateur est, tout comme un objectif normal, un élément de la liste. La première partie (la garde) se trouvera dans la variable Guard et la seconde (le corps), dans Body.

Cette procédure établit la relation : " Guard est la sous-liste d'objectifs situés avant l'objectif "!" dans la liste Clause, et Body est la sous-liste d'objectifs situés après l'objectif "!" dans la liste Clause. "

Pour la résolution de guarded_clause(Goal,Guard,Body), on recherche d'abord, via la résolution du sous-objectif find_clause, une clause dont la tête s'unifie avec Goal et puis, on détermine les deux parties (garde et corps) de la clause en se basant sur la place de l'opérateur de "trust" (grâce au sous-objectif find_guard).

Dans le cas d'une erreur syntaxique dûe à l'absence d'opérateur de "trust" dans la clause GHC, la résolution du prédicat find_guard échouera. On fera alors un

rétroparcours sur 'find_clause(Goal,Clause)' en essayant d'unifier Goal avec la clause GHC suivante.

L'absence d'analyse syntaxique telle qu'elle a été signalée à la section II.4. pose ici le problème de considérer de la même façon un échec syntaxique (absence de "trust") et un échec sémantique (unification de l'appel et de la tête de clause qui échoue pour simuler la suspension). Il serait dès lors intéressant de disposer d'une analyse syntaxique préalable à toute interprétation d'un programme GHC, de façon à ne considérer que des programmes GHC syntaxiquement corrects. Un échec de l'interprétation d'un programme GHC serait alors uniquement dû à une erreur sémantique.

La clause de la procédure correspondant à guarded_clause est dès lors la suivante

guarded_clause(Goal,Guard,Body) :- find_clause(Goal,Clause),
find_guard(Clause,Guard,Body).

Les deux sections suivantes seront consacrées respectivement au raffinement du sous-problème find_clause et à celui de find_guard.

II.5.8. Raffinement du sous-problème find_clause

Le sous-problème find_clause peut à son tour être considéré comme composé de différents sous-problèmes. Il faut en effet :

1) trouver une clause dont la tête a le même foncteur et le même nombre d'arguments que l'appel à résoudre.
2) unifier cette tête de clause avec l'appel, soit le sous-problème *unify*.
3) transformer le corps de la clause en une liste dont chaque élément correspondra à un sous-objectif, soit le sous-problème *liste*.

Le *premier sous-problème de find_clause* sera résolu par emploi de prédicats prédéfinis en Prolog séquentiel (cfr [ClMe84]) : les prédicats functor(T,F,N) et clause(X,Y).

Le prédicat **functor(T,F,N)** établit la relation : " T est une structure dont le foncteur est F et dont le nombre d'arguments est N. " La conjonction de deux appels à functor est nécessaire pour obtenir un foncteur dont les arguments sont des variables non instanciées. Cela permet de rechercher une clause sans considérer les valeurs des arguments de l'appel.

Exemple : functor(A,F,N), functor(A1,F,N) : si A = f(1,3)
alors, F = f, N = 2, A1 = f(X,Y).

La résolution du prédicat **clause(X,Y)** instancie X et Y respectivement à la tête et au corps d'une clause se trouvant dans la base de données. X doit donc être suffisamment instancié que pour pouvoir déterminer le foncteur de la tête de clause. S'il n'y a pas de clause correspondant à X, le prédicat échoue. S'il y a plus d'une clause correspondant à X, Prolog prend la première. Dans ce cas, si on essaie par la suite de resatisfaire cet objectif, les autres clauses seront choisies une à la fois. C'est ici que la similitude syntaxique entre le Prolog séquentiel et GHC est exploitée : le prédicat clause considère les clauses GHC comme des clauses Prolog. Les prédicats 'functor' seront utilisés pour pouvoir utiliser explicitement la procédure d'unification GHC définie ci-dessous. En effet, si on avait écrit directement 'clause(Goal,Clause)' sans utiliser les prédicats functor, l'unification de l'objectif Goal avec la tête de clause GHC Clause serait simplement l'unification standard de Prolog. Or, l'unification à utiliser ici doit être spécifique afin de tenir compte des règles de suspension.

Ce premier sous-problème est décrit par la conjonction d'appels suivante :

functor(Goal,Functor,Nbarg), functor(Newgoal,Functor,Nbarg),
clause(Newgoal,Clause)

Le *deuxième sous-problème de find_clause* est le sous-problème unify. Il correspond à la procédure unify(X,Y) de l'interpréteur. X est un terme de l'appel qui doit être résolu et Y est le terme correspondant dans la tête de clause GHC qui est essayée pour la résolution de l'appel. Cette procédure unifie X et Y en fonction des différents cas repris dans le tableau ci-dessous. Si X et Y peuvent être unifiés sans violer aucune règle sémantique de GHC, c'est la réussite de l'objectif unify; sinon, c'est l'échec.

terme de l'appel : X	terme de la clause : Y	résultat de l'unification
1) argument quelconque	variable non instanciée	comme Prolog : X = Y
2) variable non instanciée	argument instancié	échec (simulation de suspension)
3) constante	constante	comme Prolog : X = Y
4) liste	liste	unifier les élts 2 par 2 en se basant sur les 3 premiers cas
5) foncteur	foncteur	unifier les foncteurs et les arguments 2 par 2 en se basant sur les 3 premiers cas

Cette procédure utilise l'unification disponible en Prolog en faisant appel au prédicat prédéfini '='. La construction de la procédure unify(X,Y) est basée sur les différents cas identifiés dans le tableau.

Le premier cas est celui où Y est une variable non instanciée et X peut être indifféremment instanciée ou non. Les deux termes sont unifiés directement en utilisant l'unification du Prolog séquentiel. Il n'y a aucun problème d'exportation de valeur illicite puisque Y est non instanciée. La réussite ou non de l'unification pour GHC dépendra simplement du résultat de l'unification pour Prolog. Si celle-ci échoue, l'échec de unify(X,Y) doit être signalé car aucune autre clause unify ne peut être utilisée pour traiter ce cas, d'où la nécessité d'un "cut" avant l'appel au prédicat '='.

La clause décrivant ce cas est la suivante :

unify(X,Y) :- var(Y), !, X = Y.

Le deuxième cas est celui où Y n'est pas une variable non instanciée et X est une variable non instanciée.

Ce cas d'unification enfreint la règle de suspension (a) (cfr point c, section I.5.2.) car, à partir de la partie passive d'une clause (à partir de la tête de la clause), on voudrait lier une variable de l'appel (X) avec un terme non variable (Y), alors que la clause n'a pas franchi l'opérateur de "trust".

Il faut donc faire échouer l'unification sans possibilité de rétroparcours. Cet échec simule la suspension de l'évaluation de la clause considérée. Un appel explicite à la combinaison 'cut-fail' sera donc effectué. Ce cas est d'une importance primordiale pour le respect de la sémantique du langage GHC.

La clause correspondant à ce cas sera placée juste après celle du premier cas, de façon à n'être considérée que dans le cas où Y n'est pas une variable non instanciée. Cette clause est la suivante :

```
unify(X,Y) :- var(X), !, fail.
```

Le quatrième cas est celui où les deux arguments sont des listes. On unifie ces deux listes en unifiant les deux premiers éléments de chacune d'elles et en traitant les queues des listes de façon récursive.

On peut dans ce cas distinguer une situation générale et une condition d'arrêt.

La situation générale est celle où il y a encore au moins un élément à considérer dans les deux listes. On unifie alors les deux premiers éléments et on unifie ensuite les deux queues de listes. Aucun rétroparcours ne devant être effectué si une de ces deux unifications échoue, un "cut" est nécessaire avant les deux appels récursifs d'unification.

La condition d'arrêt est celle où il n'y a plus d'élément dans aucune des deux listes. Les deux arguments sont instanciés à la liste vide. L'unification dans ce cas réussit sans alternative possible si un rétroparcours de plus haut niveau se passe.

Deux clauses seront nécessaires à la description de ce cas :

```
unify([Elt1|List1],[Elt2|List2]) :- !, unify(Elt1,Elt2),
                                       unify(List1,List2).
unify([],[]) :- !.
```

Une solution unique sera proposée pour les troisième et cinquième cas. Ces deux cas concernent l'unification de deux types différents de structures de données :

- deux foncteurs avec leurs arguments
- deux constantes.

Considérons tout d'abord deux termes à unifier qui se présentent sous la forme de deux *foncteurs suivis de leurs arguments*.

Ils sont transformés chacun en une liste où le premier élément est le foncteur et où chaque élément suivant correspond à un argument. Cette transformation est effectuée par appel au prédicat Prolog prédéfini 'X =.. Y' qui établit la relation : " Y est la liste dont le premier élément est le foncteur de X et dont les éléments suivants sont les arguments de X. "

Les premiers éléments de chaque liste (les deux foncteurs) sont unifiés directement par le mécanisme d'unification de Prolog car ce sont deux constantes (aucun problème de suspension ne peut intervenir). Cette unification est réalisée par l'utilisation de la même variable 'F' comme premier élément des décompositions en listes de X et de Y. Dans le cas où les deux foncteurs ne sont pas les mêmes, cette unification échouerait. Les deux queues des listes sont unifiées par un appel récursif correspondant au quatrième cas : les deux arguments à unifier sont deux listes.

Le traitement du cas où les deux termes à unifier sont *deux constantes* est similaire car une constante est un foncteur sans argument. La décomposition d'une constante 'a' au moyen du prédicat Prolog' =.. ' donne comme liste résultat : ['a' | []]. Aucun problème d'exportation n'étant à considérer, l'unification directe des deux constantes se fait par l'utilisation de la même variable comme tête des deux listes de décomposition. En cas de constantes différentes, l'unification échoue. Les deux queues de listes instanciées à la liste vide sont unifiées avec succès. Cette situation correspond en effet à la condition d'arrêt du quatrième cas.

Ces deux cas d'unification de deux foncteurs avec leurs arguments et de deux constantes correspondent à la clause suivante :

```
unify(X,Y) :- X =.. [F|ArgX], Y =.. [F|ArgY], unify(ArgX,ArgY).
```

Les différents cas qui viennent d'être présentés peuvent être regroupés de façon à former le code complet de la procédure unify(X,Y).

```
unify(X,Y) :- var(Y), !, X = Y.
unify(X,Y) :- var(X), !, fail.
unify([Elt1|List1],[Elt2|List2]) :- !, unify(Elt1,Elt2),
                                       unify(List1,List2).
unify([],[]) :- !.
unify(X,Y) :-  X =.. [F|ArgX], Y =.. [F|ArgY], unify(ArgX,ArgY).
```

Le *troisième sous-problème de find_clause* est le sous-problème liste. Il concerne la tranformation d'une conjonction d'objectifs en une liste. Il correspond à la procédure liste(Terms,List).

Ce sous-problème se pose car, comme le corps de la clause candidate qui est sélectionnée se trouve sous la forme d'une conjonction de sous-objectifs, il faut le

modifier afin d'obtenir une liste où chaque élément correspond à un sous-objectif. Cette transformation est indispensable pour pouvoir examiner les objectifs un à un de façon à déterminer ceux qui appartiennent à la garde et ceux qui appartiennent au corps de la clause (la séparation garde-corps correspond au sous-problème find_guard décrit à la section suivante).

La technique de représentation de listes sous forme de listes de différence de listes est exploitée. On ne doit dès lors pas utiliser de primitive de concaténation de listes (cfr point b de la section II.5.3.).

La conjonction d'objectifs est décomposée en une liste grâce au prédicat '=..' déjà décrit ci-dessus. Cette liste ne peut néanmoins pas être utilisée telle quelle car il faut supprimer les virgules apparaissant dans la conjonction. On ne traite donc que la liste d'arguments résultant de la décomposition.

Exemple : conjonction : (a,b,c)

Conj =.. [',' | [a , (b,c)]]

La situation générale est celle où il y a encore au moins un élément de la conjonction qui n'a pas été placé dans la liste résultat. Dans ce cas, on décompose la conjonction d'objectifs restants avec le prédicat '=..' et le premier élément de la liste d'arguments est ajouté dans la liste résultat. On traite alors de façon récursive la conjonction correspondant au second élément de la liste d'arguments.

La condition d'arrêt est celle où il n'y a plus d'élément dans la conjonction. Dans ce cas, la décomposition à l'aide du prédicat '=..' échoue. La procédure liste(Terms,List) peut être décrite en Prolog de la façon suivante :

```
liste(Terms,List) :- liste(Terms,X,X,List,[]).
liste(Conj,Head,Tail,Head2,Tail2) :-
                    Conj =.. [',',H,T]], !,
                    liste(H,Head,Tail,Head1,Tail1),
                    liste(T,Head1,Tail1,Head2,Tail2).
liste(Conj,Head,[Conj|Tail],Head,Tail).
```

Les trois sous-problèmes de find_clause ayant été résolus, la procédure find_clause(Goal,Clause) peut être définie :

```
find_clause(Goal,Clauselist) :-
                        functor(Goal,Functor,Nbarg),
                        functor(Newgoal,Functor,Nbarg),
                        clause(Newgoal,Clause),
                        unify(Goal,Newgoal),
                        liste(Clause,Clauselist).
```

II.5.9. Construction de la procédure find_guard(Clause,Guard,Body)

Les éléments de la liste Clause vont être considérés un à un. Cet examen peut être caractérisé par deux situations : la situation générale en cours d'examen, et la situation finale de la condition d'arrêt.

La *situation générale* est celle où l'élément '!' n'a pas encore été rencontré. Il faut dans ce cas instancier le premier élément de la liste Guard à la valeur du premier élément de la liste Clause, avant de continuer récursivement l'examen avec pour arguments la queue de la liste Clause, la queue de la liste Guard, et la variable Body inchangée.

La *condition d'arrêt* correspond à la situation où l'élément '!' est le premier élément de la liste Clause. Il faut alors instancier la variable Body à la valeur de la queue de la liste Clause, et la liste Guard à la liste vide. Le traitement est ainsi terminé car tous les éléments précédant l'élément '!' se trouvent dans la liste Guard, et tous ceux suivant l'élément '!', dans la liste Body.

La procédure de l'interpréteur qui correspond à find_guard est dès lors la suivante

```
find_guard(['!'|Body],[],Body) :- !.
find_guard([Goal|Tail],[Goal|Guard],Body) :-
                        find_guard(Tail,Guard,Body).
```

Ce sous-problème était le dernier à considérer dans le cadre du raffinement du processus-OU. Toutes les procédures de l'interpréteur ayant été décrites, elles peuvent être assemblées pour former le code complet de l'interpréteur qui est présenté à la section suivante.

II.6. Code de l'interpréteur

```
solve(Goals) :- schedule(Goals,X,X,Head,[cycle|Tail]),
                ghc_solve(Head,Tail,échec).

schedule([],Head,Tail,Head,Tail).
schedule([Goal|List],Head,[Goal|Tail],Newhead,Newtail) :-
                        schedule(List,Head,Tail,Newhead,Newtail).

ghc_solve([cycle],[],_) :- !.
ghc_solve([cycle|Head],[cycle|Tail],pas_échec) :- !,
        ghc_solve(Head,Tail,échec).
```

```
ghc_solve([cycle|Head],Tail,échec) :- !, fail.
ghc_solve([G|Head],Tail,E) :- builtin(G),
                              call(G), !,
                              ghc_solve(Head,Tail,pas_échec).
ghc_solve([G|Head],[G|Tail],E) :- builtin(G), !,
                                  ghc_solve(Head,Tail,E).
ghc_solve([G|Head],Tail,E) :- reduce(G,R), !,
                  schedule(R,Head,Tail,Newhead,Newtail),
                  ghc_solve(Newhead,Newtail,pas_échec).
ghc_solve([G|Head],[G|Tail],E) :- ghc_solve(Head,Tail,E).

reduce(Goal,Reduced) :- guarded_clause(Goal,Guard,Reduced), solve(Guard).

guarded_clause(Goal,Guard,Body) :- find_clause(Goal,Clause),
                                   find_guard(Clause,Guard,Body).

find_clause(Goal,Clauselist) :-
                functor(Goal,Functor,Nbarg),
                functor(Newgoal,Functor,Nbarg),
                clause(Newgoal,Clause),
                unify(Goal,Newgoal),
                liste(Clause,Clauselist).

find_guard(['|'|Body],[],Body) :- !.
find_guard([Goal|Tail],[Goal|Guard],Body) :- find_guard(Tail,Guard,Body).

unify(X,Y) :- var(Y), !, X = Y.
unify(X,Y) :- var(X), !, fail.
unify([Elt1|List1],[Elt2|List2]) :- !, unify(Elt1,Elt2), unify(List1,List2).
unify([],[]) :- !.
unify(X,Y) :-  X =.. [F|ArgX], Y =.. [F|ArgY], unify(ArgX,ArgY).

liste(Terms,List) :- liste(Terms,X,X,List,[]).
liste(Conj,Head,Tail,Head2,Tail2) :-
                Conj =.. [',',H,T]], !,
                liste(H,Head,Tail,Head1,Tail1),
                liste(T,Head1,Tail1,Head2,Tail2).
liste(Conj,Head,[Conj|Tail],Head,Tail).
```

Le lecteur trouvera en annexe A le code plus volumineux de l'interpréteur augmenté d'une possibilité de trace. Cette possibilité ne modifie en rien la logique du programme de l'interpréteur telle qu'elle a été définie à la section II.5.

Un exemple y est aussi donné du résultat de la trace d'exécution du programme de fusion construit à la section I.5.3.

II.7. Modification de la stratégie d'ordonnancement

Une seconde version de la procédure schedule est proposée ici. Elle a pour but d'offrir une résolution tout-à-fait non-déterministe-ET pour les objectifs d'une conjonction (voir première version de schedule à la section II.5.4.).

Le principe est le suivant : plutôt que d'ajouter un sous-objectif en fin de file d'ordonnancement (conformément à la stratégie d'ordonnancement en largeur d'abord), le sous-objectif est ajouté à un endroit quelconque de la file. Cet endroit correspond à un nombre aléatoire généré sur base de la longueur de la file.

Le sous-objectif à ajouter dans la file d'ordonnancement est soit le résultat de la réduction d'un objectif, soit un objectif qui vient d'être suspendu et qui est ainsi reporté à un endroit quelconque de la file d'ordonnancement.

Il faut remarquer qu'une telle stratégie entraîne la perte de l'équitabilité qui était garantie avec la stratégie en largeur d'abord. Il faudrait veiller à assurer cette équitabilité par un mécanisme contrôlant le choix d'un objectif pour la résolution parmi les objectifs de la file des objectifs non encore résolus. Aucun mécanisme de ce genre n'est proposé ici.
On aurait pu considérer une insertion en position décidée par un 'démon' plutôt que de prendre une insertion en position aléatoire. Cette position décidée par le 'démon'serait telle que les échecs cycliques seraient évités et que l'équitabilité entre objectifs restant à résoudre serait garantie. Une telle technique aurait évidemment été fort complexe à implémenter.

Une première conséquence de cette modification de l'ordonnancement est que la liste des processus ordonnancés n'est plus implémentée sous la forme d'une liste de différence de listes. En effet, on n'ajoute plus les éléments simplement à la fin de la liste. Le mécanisme d'insertion d'un élément est plus compliqué, et doit faire appel à une routine d'insertion spéciale (qui n'utilise toutefois pas explicitement la primitive de concaténation de listes).

La seconde conséquence de cet ajout à une place aléatoire dans la file d'ordonnancement est plus grave. Il est maintenant impossible d'utiliser le mécanisme de détection d'échec cyclique vu au point c de la section II.5.3. car plus rien n'est séquentiel dans l'ordre de la file, on ne peut donc plus se baser sur le fait que l'élément artificiel 'cycle' se retrouve en tête de liste pour affirmer que tous les objectifs de la liste ont été examinés une fois. Une possibilité pour détecter

une situation d'échec serait de comparer, à chaque manipulation de la liste, l'état de la liste avant l'ordonnancement et l'état après ordonnancement. La lenteur de cette stratégie est la raison pour laquelle elle n'a pas été implémentée.

Voyons comment la procédure schedule peut être spécifiée et ensuite construite.

La procédure schedule(A,List,Newlist) établit la relation : " La liste Newlist est la liste correspondant à la file d'ordonnancement List dans laquelle ont été ordonnancés les sous-objectifs du système A, conformément à une stratégie d'ordonnancement aléatoire. "

La construction de la procédure est exactement la même que celle de la première version de schedule. En effet, seule doit être modifiée l'insertion d'un élément dans la file. Plutôt que d'utiliser directement les listes de différence de listes, il faut ici calculer la longueur de la file, générer un nombre aléatoire sur base de cette longueur et insérer ensuite l'élément concerné à la place suivant l'élément dont le numéro correspond au nombre aléatoire généré. Les éléments suivants sont dès lors tous déplacés d'un rang.

Cette nouvelle procédure schedule peut être codée de la façon suivante :

```
schedule([],List,List).
schedule([Goal|Tail],List,Newlist) :-
                    length(List,L), rando(L,Loc),
                    insert(Goal,List,List2,Loc).
                    schedule(Tail,List2,Newlist).
```

Le lecteur trouvera en annexe B le code de l'interpréteur implémentant la stratégie d'ordonnancement aléatoire décrite ci-dessus.

II.8. Comparaison avec le Prolog séquentiel

Il semble intéressant de faire ressortir les différences qui existent entre cette implémentation du langage GHC et le langage séquentiel Prolog.

La différence principale est l'utilisation pour GHC d'une stratégie d'ordonnancement en largeur d'abord pour la résolution d'une conjonction d'objectifs, tandis que pour Prolog, la stratégie choisie est la stratégie en profondeur d'abord. Les objectifs, en Prolog, sont donc toujours résolus dans l'ordre où ils apparaissent dans la conjonction. Comme nous l'avons vu au point a de la section II.5.3., cela n'est pas le cas pour GHC : l'ordre de résolution des objectifs est non-déterminé par

rapport à la place qu'ils occupent dans la conjonction. La stratégie utilisée pour GHC est équitable, tandis que celle utilisée pour Prolog ne l'est pas.

Une deuxième différence est la procédure d'unification de GHC qui simule une suspension par un échec quand on essaie d'exporter une liaison à l'appel. L'unification de Prolog ne simule aucune suspension car Prolog est un langage séquentiel et la suspension est nécessaire pour éviter les risques d'interférence dûs au parallélisme.

On pourrait considérer la présence d'une garde associée à toute clause GHC comme une troisième différence. Il faut toutefois remarquer que Prolog dispose aussi d'une possibilité de test pour sélectionner une clause candidate de façon définitive. En effet, l'utilisation du "cut" empêche de revenir sur un choix déjà fait pour la clause à utiliser pour réduire un objectif. Le mécanisme de rétroparcours de Prolog n'est jamais utilisé après que la résolution d'un objectif ait franchi un prédicat prédéfini "cut". La possibilité de sélectionner une clause candidate de façon définitive n'est donc pas une caractéristique propre à GHC.

II.9. Comparaison avec Concurrent Prolog

Cette comparaison avec Concurrent Prolog sera effectuée sur base de l'interpréteur de GHC proposé ici et de l'interpréteur écrit par Shapiro pour Concurrent Prolog (cfr [Sh83]).

Voyons tout d'abord en quoi la structure générale des deux interpréteurs est la même.

Le parallélisme-OU n'est simulé dans aucun des deux interpréteurs. Le non-déterminisme-ET est approché par l'emploi de la même stratégie : une stratégie d'ordonnancement en largeur d'abord. Le parallélisme tête-garde-corps propre à GHC n'est pas simulé, cela supprime donc une possibilité de différence avec la structure de l'interpréteur de Concurrent Prolog. La gestion d'environnements multiples propre à Concurrent Prolog n'est pas simulée, car l'emploi du mécanisme de rétroparcours de Prolog permet, tout comme pour GHC, de n'effectuer aucune liaison avant que la clause candidate ne soit sélectionnée. Aucun des deux interpréteurs n'effectue de vérification syntaxique du code (Concurrent Prolog ou GHC) qui doit être interprété. Signalons enfin que les deux interpréteurs exploitent l'environnement du Prolog séquentiel.

La seule différence existant entre les deux interpréteurs se situe au niveau de la primitive d'unification. En effet, la primitive de GHC ne doit pas faire de traitement spécial pour les annotations "read-only" propres à Concurrent Prolog, elle doit vérifier si on exporte une liaison à l'appel et si tel est le cas, faire échouer pour

simuler une suspension. La procédure d'unification de Concurrent Prolog doit quant à elle envisager la possibilité d'avoir des variables annotées "read-only".

Grâce à l'interpréteur ainsi développé pour le langage GHC, des applications rédigées en GHC ont pu être exécutées. Nous proposons, au chapitre suivant, la description d'une de ces applications.

Chapitre III

Application robotique en GHC

III.1. Description de l'application

Cette application est appellée "application des gels acryliques". Comme elle a déjà servi de support à un exemple dans [Va85], sa description ne reprendra que les principales caractéristiques.

L'atelier concerné par cette application est un atelier flexible d'assemblage de plaques. Un assemblage est constitué de deux plaques de verre unies à l'aide de trois pinces. Chacune des trois pinces est placée à mi-distance d'un côté des deux plaques. Le montage terminé, on obtient un récipient plat destiné à la fabrication de minces couches de gels acryliques. Le quatrième côté de l'assemblage n'est pas fixé par une pince, afin de garder une ouverture pour pouvoir verser les gels.

Cet atelier sera utilisé par le laboratoire d'analyse d'un service médical de l'hôpital Gasthuisberg de la K.U.L. (Katholieke Universiteit Leuven).

L'environnement de travail est constitué d'un distributeur de plaques, un distributeur de pinces, un site de montage et un site de dépôt.

Pour exécuter une tâche répétitive automatiquement, il est plus simple que les objets manipulés soient à une position fixe. C'est pourquoi chacun des distributeurs est doté d'un mécanisme de présentation qui présente les objets (plaques ou pinces) toujours dans la même position. De façon similaire, la station de dépôt est pourvue d'un mécanisme permettant d'entreposer un assemblage en le déposant à un endroit fixe.

Si l'application doit être exécutée par un robot unique, on peut décrire l'enchaînement des tâches à effectuer de la façon suivante. Il faut prendre une plaque, la déposer sur le site de montage, et répéter la même chose pour une seconde plaque. Il faut ensuite fixer les deux plaques en plaçant une pince aux milieux de trois des quatre côtés. Après chaque fixation de pince, l'assemblage est tourné de 90 °, de façon à ce que l'endroit d'attache d'une pince soit toujours le

même. Lorsque les trois pinces sont fixées, l'assemblage final est saisi et transféré au site de dépôt.

Il est indispensable de caractériser l'interface qui sera offert par la couche objet (cfr [Va85], [Bet86]) afin de pouvoir décrire plus précisément l'enchaînement des tâches à effectuer.

La couche objet devrait proposer un langage de programmation évolué pour la programmation de robots, langage qui offrirait à l'utilisateur la possibilité de concevoir d'une manière naturelle la solution à un problème robotique.
Le langage à implémenter devrait référencer chaque objet de l'espace de travail du robot par un nom et ainsi permettre à l'utilisateur de produire des actions sur ces objets grâce à une syntaxe simple et explicite. Les actions proposées sont au nombre de quatre :

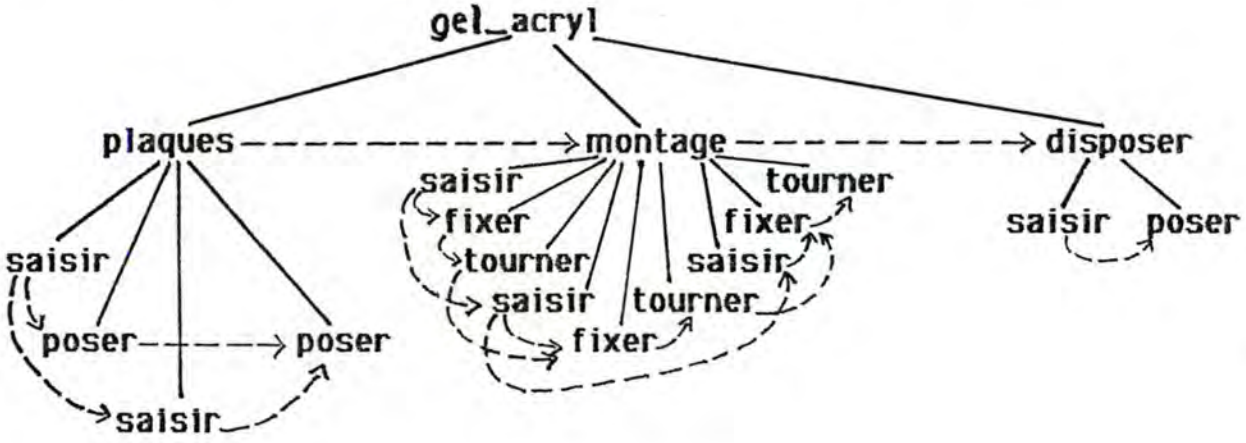
saisir, poser, fixer, tourner.

Chacune d'elles est caractérisée par deux paramètres :

saisir(**Objet,Site**) = saisir l'**Objet** se trouvant sur le **Site**
poser(**Objet,Site**) = poser l'**Objet** sur le **Site**
fixer(**Objet,Site**) = fixer l'**Objet** au **Site**
tourner(**Objet,X**) = tourner l'**Objet** de **X °**.

Sur base de ces quatre primitives et de la description de l'application, on peut considérer un arbre représentant l'enchaînement des différentes actions à effectuer pour réaliser l'assemblage pour les gels acryliques.

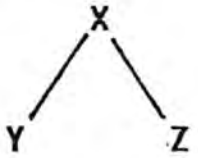
Plusieurs niveaux existent dans l'arbre, et correspondent aux niveaux que l'utilisateur considère en raffinant successivement la description.



Les feuilles de l'arbre (les primitives élémentaires pour l'utilisateur) correspondent chacune à un appel à une action primitive de la couche objet.

La relation existant entre deux niveaux successifs de l'arbre est une relation de décomposition.

exemple :



la tâche X est décomposée en deux sous-tâches : Y et Z

Les relations de précédence obligatoire entre les actions d'un même niveau de décomposition sont indiquées par des flèches entre les éléments.

ex : **X** ----> **Y** : Y ne peut commencer avant la terminaison de X.

A condition de respecter ces contraintes, les actions peuvent être effectuées en parallèle, ce qui est techniquement réalisable si plusieurs robots peuvent être utilisés simultanément.

III.2. Construction du code GHC

III.2.1. Parallélisme du code GHC

La façon d'écrire l'application en GHC ne tient aucun compte des facilités offertes par les restrictions dûes à l'interpréteur décrit au chapitre II en ce qui concerne le parallélisme-OU et le parallélisme tête-garde-corps. En effet, le code est construit de façon à tenir compte de l'existence d'un parallélisme-OU, ainsi que de celle d'un parallélisme au niveau de l'exécution de la garde et du corps d'une même clause.

Une implémentation plus parallèle de l'interpréteur ne devrait dès lors pas entraîner de modification dans le code GHC de l'application.

III.2.2. Etapes de construction du code GHC

Le programme GHC qui va être construit réalisera l'application décrite à la section III.1.

La première étape de cette construction est une description de la gestion des appareils utilisés pour exécuter les tâches de l'application. Après avoir décrit les appareils disponibles, les principales caractéristiques de leur utilisation seront abordées : priorité, exécution des primitives, pannes.

La couche objet n'étant pas encore implémentée au moment de la conception de cette application en GHC, une simulation de l'interface de cette couche a du être réalisée. La deuxième étape donnera donc un aperçu de la façon dont est simulée l'exécution des primitives par les appareils.

Les ordres à exécuter doivent être envoyés aux appareils de façon à respecter l'enchaînement des tâches tel qu'il a été défini dans la description de l'application. La dernière étape abordera donc la façon dont les tâches seront synchronisées. Un mécanisme permettant de gérer des contraintes de succession sera employé, de manière à pouvoir séquentialiser les tâches de l'application qui l'exigent.

III.3. Gestion des appareils

III.3.1. Description des appareils disponibles

Il est intéressant dans une application robotique, tout comme dans le domaine des Systèmes d'Exploitation, de classifier les appareils physiques selon des catégories conceptuelles. Cette classification permet de limiter les endroits où des modifications doivent être effectuées dans les programmes en cas d'ajout ou de suppression d'un appareil physique.

Le niveau d'abstraction ainsi introduit permet de décrire les procédures en termes d'appareils conceptuels, indépendamment du parc d'appareils effectivement disponibles.

Dans le cadre de l'application des gels acryliques, il est raisonnable de supposer qu'il y a deux catégories d'appareils conceptuels : une classe d'appareils ayant les caractéristiques de robots à cinq degrés de liberté et une classe d'appareils à un degré de liberté. Les robots à cinq degrés de liberté peuvent exécuter toutes les primitives nécessaires à la réalisation des tâches. Les appareils à un degré de liberté ne peuvent qu'exécuter la primitive 'tourner' consistant à faire tourner de X ° un objet sur lui-même. Dans le cas où les deux catégories d'appareils sont disponibles et que la primitive à exécuter est 'tourner', le choix de l'appareil à utiliser devra donc favoriser un appareil à un degré de liberté.

Les appareils physiques regroupés dans la classe conceptuelle des *robots* sont des robots similaires au robot RM 501 Movemaster de Mitsubishi (cfr [Va85]). On suppose par hypothèse que deux robots peuvent être utilisés pour exécuter les tâches de l'application , soit rob1 et rob2.

La classe conceptuelle des *tourneurs* regroupe des appareils à un degré de liberté utilisés pour faire tourner un objet sur lui-même. On suppose par hypothèse qu'un seul tourneur physique peut être utilisé pour l'exécution de l'application, soit tourn1.

III.3.2. Utilisation des appareils

a) Envoi des ordres aux appareils

Les ordres sont envoyés un à un aux appareils, et un ordre n'est envoyé qu'à un seul appareil conceptuel à la fois. Le choix de l'appareil est donc préalable à l'envoi de l'ordre.

b) Priorité des appareils conceptuels

Quand un seul type d'appareil conceptuel peut être utilisé pour une tâche, on le désigne directement. Par contre, lorsque les deux classes d'appareils (robot et tourneur) peuvent être utilisées indifféremment pour une tâche donnée, il faut se baser sur leur priorité afin de déterminer quelle classe sera chargée d'exécuter la primitive.

Les priorités des appareils conceptuels sont fixes pour une application donnée car pour chaque classe d'appareils, on considère leurs possibilités techniques (qui restent inchangées durant l'exécution), et pas des critères tels que la proximité par rapport à l'objet à manipuler (varie au cours de l'exécution de l'application). Pour les tâches pouvant être indifféremment exécutées par des appareils appartenant aux deux classes conceptuelles, la plus grande priorité a été assignée à tourneur, robot ne sera donc choisi que si l'appareil physique de la catégorie tourneur est indisponible.

Nous allons voir comment sont réalisées les descriptions de primitives pouvant être exécutées par les deux classes d'appareils, ainsi que celles de primitives pour l'exécution desquelles l'appareil conceptuel pouvant être utilisé est unique.

◇ Pour l'application des gels acryliques, la seule *primitive pouvant être exécutée par les deux classes d'appareils* est la primitive 'tourner'.

La procédure correspondant à cette primitive 'tourner' effectue le choix d'un appareil conceptuel et l'envoi de l'ordre d'exécution à l'appareil ainsi choisi.

Deux arguments sont nécessaires pour décrire l'objet qui doit être tourné (Objet), et le nombre de degrés de rotation de l'objet (Degré). Deux autres arguments seront nécessités par la synchronisation des tâches (X et Y). Leur utilisation sera décrite à la section III.5.

Comme la sélection de l'appareil utilisé doit absolument précéder l'envoi de l'ordre d'exécution de la tâche à l'appareil, il était impératif que la sélection corresponde à un prédicat (prio_concept(X)) situé dans la garde de la clause de la

procédure, et que l'envoi de l'ordre (appareil_concept(X,T,Y)) soit situé dans le corps. Cependant, nous avons vu à la section III.2.1. que le code GHC devait être construit de façon à pouvoir tenir compte d'un parallélisme entre l'évaluation de la garde d'une clause et l'exécution du corps de celle-ci. De façon à tenir compte de cette possibilité et à séquentialiser le choix de l'appareil et l'envoi de l'ordre d'exécution, les deux prédicats prio_concept et appareil_concept partageront une variable X. La valeur de X identifiera l'appareil choisi devant être utilisé pour l'exécution. Cette variable correspondra, dans la procédure appareil_concept, à un argument instancié dans les têtes de clauses. Tant que la variable X n'aura pas été instanciée par le prédicat prio_concept, l'exécution du prédicat appareil_concept sera suspendue. En effet, si tel n'était pas le cas, on exporterait la valeur se trouvant dans la tête de clause de appareil_concept à la variable X non instanciée dans l'appel. De cette façon, l'exécution du corps de la clause, même si elle débute avant celle de la garde, ne pourra avoir lieu que quand l'appareil conceptuel à utiliser sera connu (quand la variable X sera instanciée par le prédicat prio_concept).

La procédure correspondant au prédicat *prio_concept(X)* effectue le choix de l'appareil conceptuel (robot ou tourneur) en fonction des disponibilités des appareils, et de leur priorité. Ce choix est signifié par instanciation de son unique argument X à la valeur du nom de l'appareil sélectionné.

Comme c'est la classe d'appareils 'tourneur' qui est la plus prioritaire, elle sera sélectionnée si l'appareil physique la représentant est disponible (c'est-à-dire en service (pas en panne) et libre (pas occupé)). La classe conceptuelle 'robot' ne sera choisie que si la classe 'tourneur' ne peut être utilisée (car l'appareil physique n'est pas disponible).

La vérification de disponibilité des appareils est effectuée par un appel au prédicat disponible(X). Ce prédicat est un prédicat prédéfini. Le code de la procédure lui correspondant est donc en Prolog séquentiel. Ce prédicat établit la relation : l'appareil X est en service (n'est pas en panne) et est libre (n'est pas occupé à l'exécution d'une tâche). Si une de ces deux conditions (ou les deux) n'est(ne sont) pas vérifiée(s), la résolution du prédicat échoue.

L'instanciation de l'argument X à la valeur de l'appareil choisi sera effectuée par appel au prédicat prédéfini d'unification '=' dans le corps des clauses de la procédure 'prio_concept(X)'.

Cette procédure sera dès lors codée de la manière suivante :

```
prio_concept(X) :- disponible(tourn1), !, X = tourneur.
prio_concept(X) :- not(disponible(tourn1)), !, X = robot.
```

La procédure correspondant au prédicat *appareil_concept(X,T,Y)* réalise la fonction d'envoyer l'ordre d'exécution de la tâche T à la classe conceptuelle d'appareils X. Cette procédure comportera trois arguments : le premier (X)

représente l'appareil conceptuel auquel il faut envoyer l'ordre d'exécution, le deuxième (T) contient la description de la tâche à effectuer, et le troisième (Y) est une variable de synchronisation dont le rôle sera expliqué à la section III.5.

Nous avons vu ci-dessus que l'argument correspondant à l'appareil conceptuel devait correspondre à une variable instanciée dans les têtes de clauses de la procédure. Une séquentialisation de l'évaluation de la garde et de l'exécution du corps de la clause de la procédure 'tourner' peut ainsi être effectuée. Les deux valeurs auxquelles cet argument sera instancié correspondent aux deux classes conceptuelles 'robot' et 'tourneur'. Les deux têtes de clauses peuvent être unifiées en parallèle-OU car pour une certaine valeur de classe conceptuelle, une seule des deux unifications aboutira.

L'envoi de l'ordre d'exécution est réalisé par un appel au prédicat 'robot' pour la clause dont l'argument de la tête a la valeur 'robot', et au prédicat 'tourneur' pour l'autre clause. Ces deux prédicats effectuent la sélection d'un appareil physique appartenant à la classe conceptuelle qu'ils représentent et transmettent l'ordre d'exécution à l'appareil ainsi choisi. Ils seront décrits en détails au point suivant.

La procédure appareil_concept ainsi décrite peut être codée de la façon suivante :

```
appareil_concept(robot,Tâche,Y) :- true, !, robot(Tâche,R,Y).
appareil_concept(tourneur,Tâche,Y) :- true, !, tourneur(Tâche,R,Y).
```

Les deux procédures prio_concept et appareil_concept ayant été définies, nous donnons ci-dessous le code de la procédure 'tourner' :

```
tourner(Objet,Degré,X,Y) :- prio_concept(X), !,
                            appareil_concept(X,[tourner,Objet,Degré],Y).
```

◇ Les *primitives qui ne peuvent être exécutées que par une seule classe conceptuelle d'appareils* envoient l'ordre d'exécution de la tâche directement à la classe conceptuelle. En effet, celle-ci étant imposée, aucun choix préalable ne doit être effectué. A titre d'exemple, la primitive 'saisir' va être décrite. Les autres primitives pour l'exécution desquelles la classe conceptuelle d'appareils à utiliser est imposée (poser et fixer) lui sont tout-à-fait similaires.

La primitive 'saisir' ne peut être exécutée que par un appareil de la classe 'robot'. Son exécution a pour effet de saisir un objet situé à un endroit déterminé. La procédure lui correspondant envoie, tout comme dans le cas de la procédure appareil_concept, directement l'ordre d'exécution à la classe conceptuelle robot. C'est par un appel au prédicat robot que cet envoi est effectué.

Les arguments de la procédure 'saisir' sont au nombre de cinq : l'objet à saisir, le site où l'objet se trouve, un argument destiné au choix de l'appareil physique (voir point suivant), et deux arguments utilisés pour la synchronisation de tâches (voir section III.5.).

Le code de la clause correspondant à la procédure 'saisir' est le suivant :

saisir(Objet,Site,R,X,Y) :- true, !, robot([saisir,Objet,Site],R,Y).

c) Priorité des appareils physiques

La priorité des appareils physiques est dynamique car elle est basée sur l'emplacement de l'appareil par rapport aux objets à manipuler et sur d'autres critères dépendant de l'exécution en cours (ex. : fréquence d'utilisation d'un appareil depuis le début de l'exécution).

Comme tout appareil physique n'appartient qu'à une seule classe conceptuelle, les priorités entre appareils physiques ne concernent que les appareils de la même classe. En effet, comme nous l'avons vu au point précédent, la sélection de la classe conceptuelle est effectuée avant que ne soit envisagée la distribution des tâches aux appareils physiques.

Dans certains cas, la priorité entre appareils physiques n'intervient pas.

Le premier cas est la situation où un seul appareil physique existe pour une certaine catégorie conceptuelle. Le choix de la classe conceptuelle va donc de pair avec le choix de l'appareil physique. C'est précisément le cas qui se présente pour la catégorie 'tourneur'. Comme il n'y a qu'un seul appareil physique qui puisse être utilisé (tourn1), aucun problème de priorité ne se pose.

Le second cas se présente quand on impose l'utilisation d'un appareil physique particulier. On peut, comme illustration, citer la contrainte selon laquelle le robot physique qui saisit une plaque doit être le même que celui qui pose cette plaque.

Exemple : - L'appareil **rob1** est utilisé pour saisir une plaque. La tâche suivante à effectuer est de poser cette plaque.
- Même si l'appareil **rob2** est disponible, il ne peut être utilisé pour poser la plaque, puisque c'est **rob1** qui la détient. Il faut donc que ce soit aussi **rob1** qui effectue l'action de poser la plaque.

Pour la primitive 'poser', on imposera donc le choix de l'appareil physique comme étant celui qui a été utilisé pour la primitive 'saisir'. Il en sera de même pour la primitive 'fixer' : l'appareil utilisé pour exécuter cette primitive devra être le même que celui ayant exécuté la primitive 'saisir'.

Lorsque le choix de l'appareil physique d'une classe conceptuelle n'est pas imposé, ce sont les priorités qui interviennent. Dans le cadre de l'application des gels acryliques, les priorités ne peuvent jouer qu'entre les deux appareils rob1 et rob2 de la catégorie conceptuelle robot.

Comme nous venons de le signaler, les priorités d'appareils physiques sont propres à chaque classe conceptuelle. Il sera donc nécessaire de définir une procédure par classe conceptuelle. Ces deux procédures correspondent aux prédicats 'robot' et 'tourneur' décrits sommairement au point précédent. La rôle des deux procédures est le même : sélectionner un appareil physique appartenant à la classe conceptuelle concernée et envoyer l'ordre d'exécution de la tâche à cet appareil. Ces procédures auront trois arguments : la description de la tâche à exécuter (Tâche), un appareil physique éventuellement imposé (Rob) (pour saisir-poser et saisir-fixer), et une variable de synchronisation (Y) (cfr section III.5.). Nous allons considérer séparément les deux classes conceptuelles.

La *classe conceptuelle tourneur* ne nécessite pas une gestion de priorités entre appareils physiques car elle comprend un seul appareil physique. Le choix de celui-ci est donc immédiat. L'argument correspondant à un éventuel appareil imposé ne sera pas utilisé car les primitives saisir, poser et fixer ne peuvent être exécutées par un appareil de la classe tourneur.

L'envoi de l'ordre d'exécution de la tâche se fera dans le corps de l'unique clause de la procédure 'tourneur', par un appel au prédicat 'tourn1'. Ce prédicat effectue le lien avec les primitives de la couche objet. En effet, il appelle la primitive 'exec' qui simule l'exécution d'une tâche par un appareil physique (ici, tourn1). La description du prédicat 'tourn1' sera effectuée au point d ci-dessous.

Le code de la clause de la procédure 'tourneur' peut être facilement déduit à partir de la description qui vient d'en être faite :

tourneur(Tâche,_,Y) :- true, !, tourn1(Tâche,_,Y).

La *classe conceptuelle robot* est caractérisée par la procédure 'robot'. Trois cas de gestion de priorité doivent être envisagés :

1er cas : l'argument Rob correspondant à l'appareil physique imposé est instancié dans l'appel à la valeur 'rob1'. Nous nous trouvons dans la situation où l'exécution d'une primitive 'poser' ou 'fixer' est demandée, alors que la primitive 'saisir' préalable a été exécutée avec l'appareil physique rob1. Ce cas est traité par une clause dans la tête de laquelle l'argument Rob est instancié à la valeur rob1.

Dans ce cas, tout comme pour la clause de la procédure 'tourneur', le problème de choix de l'appareil ne se pose pas. L'envoi de l'ordre d'exécution de la tâche pourra se faire sans test supplémentaire à l'unification de la tête de clause.

Cet envoi sera effectué par un appel au prédicat 'rob1' dans le corps de la clause de 'robot'. Ce prédicat 'rob1' effectue, tout comme le prédicat 'tourn1', le lien avec les primitives de la couche objet. Il réalise la demande de simulation de l'exécution par rob1 de la tâche décrite dans le premier argument de 'robot' (Tâche). Il sera décrit au point d ci-dessous.

La clause décrivant ce premier cas est la suivante :

robot(Tâche,rob1,Y) :- true, !, rob1(Tâche,_,Y).

2ème cas : l'argument correspondant à l'appareil physique imposé est instancié dans l'appel à la valeur 'rob2'. La primitive 'saisir' a donc été exécutée par l'appareil physique rob2. De façon à identifier ce cas, la tête de la clause correspondante dans la procédure robot aura l'argument Rob instancié à la valeur rob2. Ce cas est tout-à-fait similaire au premier cas.

Un appel au prédicat 'rob2' sera effectué dans le corps de la clause. Ce prédicat 'rob2' a la même fonction que 'rob1', mais demande ici la simulation de l'exécution de la tâche Tâche par l'appareil rob2.

La clause correspondant à ce cas est la suivante :

robot(Tâche,rob2,Y) :- true, !, rob2(Tâche,_,Y).

3ème cas : l'argument correspondant à l'appareil physique imposé est une variable non instanciée dans l'appel. Nous nous trouvons dans le cas où une demande d'exécution de la primitive 'saisir' est effectuée. L'appareil physique qui sera utilisé n'est pas imposé. Il faudra cependant instancier l'argument Rob à la valeur du nom de l'appareil qui a été utilisé, de façon à pouvoir imposer ce même appareil pour la primitive (poser ou fixer) qui suivra. Le choix de l'appareil sera effectué en fonction des priorités que les appareils physiques ont à ce moment et en fonction de leur disponibilité.

La présence dans la tête de clause d'une variable non instanciée à la place de l'argument Rob n'est pas suffisante pour l'identification de ce cas. En effet, l'évaluation des trois clauses de la procédure 'robot' peut être effectuée en parallèle-OU. La clause traitant ce troisième cas ne peut être candidate que pour ce cas-ci, elle ne peut être utilisée dans les deux cas où l'argument Rob est instancié. Cependant, si on ne fait pas de test supplémentaire dans la garde de cette clause, elle pourrait être utilisée aussi dans le cas où Rob est instancié car l'unification de l'appel (avec variable instanciée) et de la tête de clause (avec variable non instanciée) réussirait.

De façon à éviter cette réussite non désirée, un appel au prédicat var(X) sera effectué dans la garde de la clause traitant ce troisième cas. Le prédicat var(X) réussit si son argument est une variable non instanciée; sinon, il échoue. L'argument X avec lequel ce prédicat sera invoqué est l'argument de l'appareil imposé (Rob).

Cette clause ne pourra donc être candidate que si l'argument de l'appareil imposé est une variable non instanciée dans l'appel.

Tout comme dans le cas de la sélection d'un appareil conceptuel (procédure tourner(Objet,Degré,X,Y)) au point b ci-dessus, l'ordre d'exécution de la tâche ne peut être envoyé avant que la sélection de l'appareil ne soit terminée. Une façon de séquentialiser l'exécution de ces deux actions est de placer l'appel au prédicat choisissant l'appareil physique (prio_phys(A,B,X)) dans la garde de la clause, et l'appel au prédicat envoyant l'ordre d'exécution (appareil_phys(X,T,Rob,Y)) dans le corps. Cependant, le corps et la garde peuvent être résolus en parallèle. Il faut donc imposer une séquentialisation par l'existence d'une variable partagée par les prédicats prio_phys et appareil_phys (soit X). Cette variable sera instanciée par le prédicat prio_phys à la valeur de l'appareil physique sélectionné. L'argument correspondant à cette variable sera instancié dans les têtes de clauses de la procédure appareil_phys. De cette façon, le consommateur appareil_phys sera suspendu tant que la variable partagée n'aura pas été instanciée par le producteur prio_phys. L'ordre ne sera dès lors envoyé que lorsque l'appareil à utiliser aura été identifié.

Nous allons raffiner les sous-problèmes correspondant aux deux prédicats cités ci-dessus : prio_phys et appareil_phys.

Le prédicat *prio_phys* a trois arguments : les deux premiers (A et B)correspondent aux deux appareils entre lesquels le choix doit être effectué, et le troisième (X) est instancié à l'appareil sélectionné parmi des deux premiers.

La sélection de l'appareil est effectuée par application des trois principes suivants :

(1) Si un seul des deux appareils est disponible, il est l'appareil sélectionné.

(2) Si les deux appareils sont disponibles, il faut consulter la base de données décrivant l'état et les priorités des appareils à tout moment de l'exécution de l'application, de façon à connaître l'appareil le plus prioritaire. C'est celui-ci qui sera l'appareil sélectionné X.

(3) Si aucun des deux appareils n'est disponible, la sélection échoue.

La sélection de l'appareil est effectuée en deux étapes : identification de la situation (quel principe appliquer ?) et instanciation de l'argument à la valeur de l'appareil sélectionné en fonction de la situation et du principe appliqué. Nous avons vu que toute exportation de liaison à l'appel devait être explicitement effectuée dans le corps d'une clause GHC, par utilisation du prédicat '=', afin de respecter la règle de suspension (a) (cfr point c, section I.5.2.). L'instanciation de l'argument X à la valeur de l'appareil sélectionné sera donc effectuée de cette façon.

La vérification de la disponibilité d'un appareil est effectuée par un appel au prédicat disponible(X). Nous avons vu que ce prédicat réussit si l'appareil X est en service et est libre.

Les vérifications de disponibilité seront effectuées dans la garde des clauses car elles font partie des critères de sélection de la clause candidate.

Le principe **(1)** regroupe deux cas :

1.1) Le premier appareil (A) est disponible, et le second (B) ne l'est pas. L'unification des arguments X et A sera donc effectuée.

1.2) Le second appareil (B) est disponible, et le premier (A) ne l'est pas. L'unification des arguments X et A sera donc effectuée.

Ces deux cas correspondront à deux clauses de la procédure prio_phys.

Le principe **(2)** regroupe également deux cas :

2.1) Les deux appareils sont disponibles et le premier appareil (A) est plus prioritaire que le second (B). L'argument X sera unifié à A.

2.2) Les deux appareils sont disponibles et le second appareil (B) est plus prioritaire que le premier (A). L'argument X sera unifié à B.

Ces deux cas nécessitent la consultation de la base de données pour connaître la priorité des appareils. Cette consultation est simulée par l'emploi du prédicat prédéfini 'MYESNO'. Ce prédicat affiche une question à l'écran et ne réussit que si l'utilisateur répond 'YES'. Les questions posées seront tout naturellement :

"premier appareil plus prioritaire que second ? "

"second appareil plus prioritaire que premier ? "

Cette simulation de consultation, tout comme les vérifications de disponibilité, seront effectuées dans les gardes des deux clauses traitant ces deux cas.

Le principe **(3)** ne nécessite pas de clause. En effet, si aucune des quatre clauses réalisant les deux premiers principes n'est candidate, cela signifie qu'aucun des deux appareils n'est disponible. Dans ce cas, la résolution du prédicat prio_phys échoue.

La façon dont les différents cas sont traduits en GHC permet d'effectuer une évaluation en parallèle-OU des quatre clauses de la procédure, sans remettre en question la réalisation des trois principes énoncés ci-dessus.

Le code de la procédure prio_phys peut maintenant être donné.

```
prio_phys(A,B,X) :- disponible(A), not(disponible(B)), !,
                    X = A.
prio_phys(A,B,X) :- disponible(B), not(disponible(A)), !,
                    X = B.
prio_phys(A,B,X) :- disponible(A), disponible(B),
                    'MYESNO'([A,plus,prioritaire,que,B]), !,
                    X = A.
prio_phys(A,B,X) :- disponible(A), disponible(B),
                    'MYESNO'([B,plus,prioritaire,que,A]), !,
                    X = B.
```

Le prédicat *appareil_phys* aura quatre arguments : le premier (X) correspond à la variable partagée avec le prédicat prio_phys dont l'utilisation a été décrite ci-dessus, le deuxième (T) est la description de la tâche à exécuter, le troisième (Rob) est la variable qui sera éventuellement instanciée au nom de l'appareil imposé, le quatrième (Y) est une variable de synchronisation de tâches.

Dans le cas de l'exécution de la primitive saisir, il faut garder le nom de l'appareil physique qui a été utilisé, de façon à pouvoir prescrire l'utilisation de ce même appareil pour la primitive poser ou fixer qui suivra. L'argument Rob sera dans ce cas instancié au nom de l'appareil chargé d'exécuter la tâche saisir.

La fonction du prédicat appareil_phys est d'envoyer l'ordre d'exécution de la tâche T à l'appareil désigné par le premier argument X (rob1 ou rob2). Cet ordre est envoyé de façon similaire à ce qui se passe pour le prédicat 'robot' : par appel au prédicat rob1 ou rob2. Ces deux prédicats effectuent le lien avec les primitives de la couche objet. Ils sont décrits au point d ci-dessous.

Deux cas doivent être envisagés :

1) X est instancié à 'rob1'. Il faut alors appeller le prédicat rob1.

2) X est instancié à 'rob2'. Il faut alors appeller le prédicat rob2.

La différenciation de ces deux cas se fait grâce à l'emploi de deux clauses dans les têtes desquelles l'argument X sera respectivement instancié à rob1 et à rob2.

Le code de la procédure appareil_phys est dès lors le suivant :

```
appareil_phys(rob1,T,Rob,Y) :- true, !, rob1(T,Rob,Y).
appareil_phys(rob2,T,Rob,Y) :- true, !, rob2(T,Rob,Y).
```

Les deux sous-problèmes représentés par les prédicats prio_phys et appareil_phys étant maintenant raffinés, nous pouvons rappeller le code des deux premières clauses de la procédure robot, et donner celui de la troisième clause, de façon à avoir le code complet de cette procédure.

```
robot(Tâche,rob1,Y) :- true, !, rob1(Tâche,R,Y).
robot(Tâche,rob2,Y) :- true, !, rob2(Tâche,R,Y).
robot(Tâche,Rob,Y) :- var(Rob), prio_phys(rob1,rob2,X), !,
                                    appareil_phys(X,Tâche,Rob,Y).
```

d) Lien avec les primitives de la couche objet

Le lien entre la description de l'application en langage GHC et les primitives de la couche objet est réalisé par les prédicats par lesquels chaque appareil physique existant peut envoyer un ordre demandant la simulation de l'exécution d'une tâche.

Les appareils physiques considérés ici étant au nombre de trois, il nous faudra décrire trois prédicats : rob1, rob2 et tourn1.

Chacun de ces prédicats aura trois arguments : le premier est la description de la tâche à exécuter (T), le deuxième est la variable qui doit être instanciée au nom de l'appareil physique qui a été utilisé (Rob) (nécessaire dans le cas de l'exécution de la tâche 'saisir', cfr point c ci-dessus) et le troisième est une variable de synchronisation (Y).

La résolution de ces trois prédicats nécessite l'appel à la primitive simulant l'exécution d'une tâche par un appareil. La primitive est nommée 'exec' et est décrite à la section III.4. L'appel à cette procédure est effectué en lui communiquant comme arguments l'identité de l'appareil physique utilisé (tourn1, rob1 ou rob2), la description de la tâche (T) et la variable de synchronisation (Y). Cette primitive simule l'exécution de la tâche par l'appareil et instancie la variable de synchronisation (voir section III.5.).

Le prédicat *tourn1* ne considèrera pas l'argument Rob. En effet, celui-ci ne doit être utilisé que dans le cas de l'exécution de la tâche saisir or, cette tâche ne peut être exécutée par l'appareil physique tourn1. La fonction de ce prédicat se réduit donc à l'appel de la primitive 'exec'.
La clause lui correspondant est la suivante :

tourn1(T,_,Y) :- true, !, exec(tourn1,_,Y).

Le prédicat *rob1*, quant à lui, doit effectuer l'instanciation de Rob à la valeur 'rob1'. Cette instanciation se fera dans le corps de la clause, par appel au prédicat '='. Tout comme pour le prédicat 'tourn1', l'appel de la primitive 'exec' sera aussi effectué dans le corps de la clause.
Le code de la clause est donc :

rob1(T,Rob,Y) :- true, !, exec(rob1,T,Y), Rob = rob1.

Le prédicat *rob2* est décrit par une clause tout-à-fait similaire à celle du prédicat rob1 :

rob2(T,Rob,Y) :- true, !, exec(rob2,T,Y), Rob = rob2.

e) Pannes d'appareils

Au début de toute application, une vérification est effectuée pour les appareils physiques susceptibles d'être employés, de façon à initialiser la description de leur état. Ils doivent alors normalement tous être en service et libres (=disponibles). La

base de données concernant les appareils physiques reprend donc pour chacun d'eux deux caractéristiques de leur état :

1) en_service ou en_panne,
2) libre ou occupé.

Nous avons vu que lorsque le choix de l'appareil physique n'est pas imposé par une utilisation antérieure (cfr saisir-poser et saisir-fixer), la primitive chargée de déterminer quel appareil sera utilisé vérifie si les appareils sont libres ou occupés, en service ou en panne.

L'état de l'appareil 'libre ou occupé' est mis-à-jour lors de la simulation de l'exécution d'une tâche par un appareil. Dans ce cas, l'appareil qui était libre de vient occupé. A la fin de l'exécution de la tâche, l'appareil occupé redevient libre. Plus de détails à ce sujet seront donnés à la section III.4.

L'état de l'appareil 'en_service ou en_panne' est automatiquement mis-à-jour par des tests réguliers. Si, lors d'un test, l'appareil ne répond pas, la description de son état passe immédiatement de 'en_service' à 'en_panne'. Puisqu'avant toute utilisation d'un appareil physique une vérification est effectuée pour voir si l'appareil est 'en_service', un appareil en panne ne pourra jamais être sélectionné.

S'il n'est pas possible de trouver un appareil physique disponible (c'est-à-dire en_service et libre) pour une certaine catégorie conceptuelle, ce sera l'échec de la tâche. Or, comme dans l'interpréteur décrit au chapitre II aucune distinction n'est faite entre échec et suspension, l'objectif ayant échoué est replacé dans la file d'ordonnancement des objectifs non encore résolus. Si les appareils étaient simplement tous occupés, la terminaison d'une des autres tâches en cours permettra à la tâche concernée d'obtenir un appareil libre pour son exécution. Si par contre tous les appareils étaient en panne, un échec cyclique serait détecté pour la tâche (cfr point c section II.5.3.) et dans ce cas, ce serait l'échec définitif de l'application.

III.4. Simulation de l'interface offert par la couche objet

La couche objet (cfr [Va85], [Bet86]) n'étant pas encore implémentée, il a fallu simuler l'interface en décrivant les primitives qui devraient normalement être disponibles (cfr section III.1.).

La simulation de l'exécution d'une primitive par un appareil se fait en affichant à l'écran un libellé correspondant à la description complète de la primitive et au nom de l'appareil utilisé.

La simulation de la mise-à-jour de la base de données décrivant l'état des appareils doit être effectuée en fonction des occupations et libérations des

appareils physiques utilisés. Deux procédures seront à cette fin nécessaires : occupation d'un appareil et libération d'un appareil.

Elles effectuent toutes deux une vérification de l'état de service de l'appareil avant de faire toute modification. Si l'appareil est en service (pas en panne), la mise-à-jour est effectuée normalement : dans le cas d'une occupation, l'appareil libre passe dans l'état occupé et dans le cas d'une libération, l'appareil occupé passe dans l'état libre. Si par contre l'appareil est en panne, aucune modification n'est effectuée et c'est l'échec de la primitive et dès lors l'échec de la simulation de l'exécution de la tâche.

Les deux procédures d'occupation et de libération ne sont pas invoquées de la même façon pour les quatre primitives. Seule la primitive tourner doit prescrire une occupation et une libération de l'appareil employé. En effet, ainsi qu'il a été expliqué au point b de la section III.3.2., l'appareil utilisé pour fixer ou pour poser un objet qui vient d'être saisi doit être le même que celui qui l'a saisi. Il ne faut donc pas libérer l'appareil après avoir saisi l'objet. De façon symétrique, aucune occupation n'est effectuée avant d'utiliser un appareil pour poser ou pour fixer. L'appareil concerné reste donc occupé du début de saisir à la fin de poser (ou fixer). Dans ce cas, une vérification de l'état de l'appareil est toutefois effectuée entre les deux tâches pour s'assurer que l'appareil est toujours en service. Une panne après l'exécution de la tâche saisir serait ainsi détectée et l'échec serait signalé.

- C'est au niveau de l'exécution des primitives que sont instanciées les variables de synchronisation. La valeur à laquelle une telle variable est instanciée dépendra de la primitive exécutée : pour la primitive saisir, ce sera la valeur *fin_saisir*; pour fixer, *fin_fixer*; pour poser, *fin_poser* et pour tourner, *fin_tourner* (voir synchronisation à la section III.5.)

La procédure réalisant ces simulations est la procédure 'exec'. Ses trois arguments sont respectivement : l'identité de l'appareil physique avec lequel la primitive doit être exécutée, la description complète de la primitive, la variable de synchronisation.

Cette primitive a été rédigée en Prolog séquentiel car elle ne fait pas partie en tant que telle de la description en GHC de l'application. Elle correspond donc à un prédicat prédéfini pour GHC.

Les occupations et libérations d'appareils étant effectuées différemment en fonction des primitives à exécuter, tout comme les instanciations de la variable de synchronisation, chaque primitive correspondra à un cas particulier, et sera décrite par une clause de la procédure 'exec'.

Les prédicats 'occupation' et 'libération' seront présents ou absents du corps des clauses en fonction des règles énoncées ci-dessus. L'affichage de la description de la primitive et de l'appareil utilisé aura lieu pour toute primitive, ainsi que l'instanciation de la variable de synchronisation. De façon à ce que l'instanciation n'ait lieu que lorsque la primitive est exécutée, le prédicat '=' sera le dernier

objectif à droite dans le corps des clauses. L'ordre d'évaluation des objectifs en Prolog est tel que ce prédicat sera bien le dernier à être résolu.

La procédure exec est codée en Prolog séquentiel de la façon suivante :

```
exec(X,[saisir,Objet,Site],Y) :- occuper(X),
                                 'PP'(saisir,Objet,en,Site,avec,X),
                                 Y = fin_saisir.
exec(X,[fixer,Objet,Site],Y) :- état(X,en_service),
                                 'PP'(fixer,Objet,en,Site,avec,X),
                                 libérer(X), Y = fin_fixer.
exec(X,[poser,Objet,Site],Y) :- état(X,en_service),
                                 'PP'(poser,Objet,en,Site,avec,X),
                                 libérer(X), Y = fin_poser.
exec(X,[tourner,Objet],Y) :- occuper(X),
                             'PP'(tourner,Objet,avec,X),
                             libérer(X), Y = fin_tourner.
```

III.5. Synchronisation des sous-tâches

III.5.1. Principe de base

Comme nous l'avons vu à la section III.1., certaines sous-tâches peuvent être exécutées en parallèle, mais les contraintes de succession doivent absolument être respectées : les exécutions des sous-tâches liées par une relation de précédence dans l'arbre doivent être séquentialisées.

Le langage GHC offrant la possibilité d'une exécution tout-à-fait parallèle, il faut trouver un moyen de contraindre certains enchaînements séquentiels de processus. Cette synchronisation de processus correspondant aux sous-tâches peut être effectuée grâce à l'exploitation des règles de suspension vues dans le cadre de la sémantique procédurale de GHC (cfr point c, section I.5.2.).

Tout couple de processus qui doivent s'enchaîner de façon séquentielle doit avoir une variable en commun afin de pouvoir communiquer pour se synchroniser.

Le *principe* selon lequel cette synchronisation est effectuée est que la tâche qui doit précéder l'autre dans la contrainte de succession considérée sera producteur pour la variable partagée, tandis que l'autre tâche en sera consommateur.

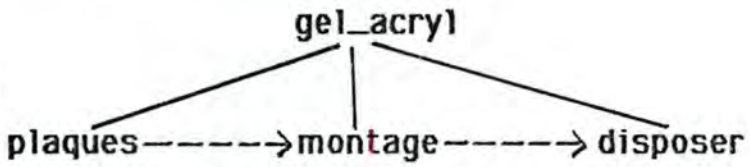
L'argument de la tête de la clause décrivant la *tâche producteur* qui correspond à la variable partagée sera non instancié. L'instanciation de cette variable aura lieu lors de la résolution du corps de la clause. Nous avons vu lors de la description des procédures des sections précédentes que c'est dans le corps des clauses que se

trouvent les appels aux primitives de l'interface de la couche objet. Ce sont ces primitives qui, en fin d'exécution, instancient les variables de synchronisation à une certaine valeur. La tâche producteur n'assignera donc une valeur à la variable partagée que lorsque son exécution sera terminée.

L'argument de la tête de la clause décrivant la *tâche consommateur* qui correspond à la variable partagée sera instancié à la valeur que le producteur doit assigner à cette variable. La tâche consommateur sera donc suspendue tant que la variable partagée n'aura pas été instanciée par le producteur. En effet dans ce cas, l'argument de l'appel non instancié devrait être unifié à la variable instanciée dans la tête de clause, ce qui serait à l'origine d'une exportation de liaison à l'appel.

Ce mécanisme de synchronisation assure donc que toute tâche producteur sera terminée avant que la(les) tâche(s) consommateur(s) ne commence(nt).

Une illustration de ce mécanisme va maintenant être donnée. Soit l'enchaînement des sous-tâches prescrit par l'arbre suivant (extrait de l'arbre complet décrivant l'application à la section III.1.) :



Les trois sous-tâches formant la décomposition de la tâche 'gel_acryl' doivent être synchronisées de façon à ce que la sous-tâche 'plaques' précède la sous-tâche 'montage' et que celle-ci précède la sous-tâche 'disposer'. Les sous-tâches 'plaques' et 'montage' devront se partager une variable (soit P), ainsi que 'montage' et 'disposer' (soit M). Les sous-tâches 'plaques' et 'montage' seront respectivement producteurs de P et de M, tandis que 'montage' et 'disposer' seront respectivement consommateurs de P et M. La sous-tâches 'disposer' sera producteur d'une variable (soit D) de façon à pouvoir être synchronisée de la même façon dans le cas où une tâche la suivant devrait être ajoutée à la description de 'gel_acryl'.

La variable partagée par 'plaques' et 'montage' est instanciée à 'fin_poser' dans le corps de la clause correspondant au prédicat 'plaques'. Cette même variable est sous la forme de la constante 'fin_poser' dans la tête de la clause de 'montage'. L'unification de l'appel *montage(P,M)* et de la tête de clause *montage(fin_poser,Y)* sera suspendue jusqu'au moment où la variable P sera instanciée à 'fin_poser' par la tâche 'plaques'. A ce moment seulement, la clause de 'montage' pourra être utilisée pour résoudre d'appel.

Le processus 'montage' devra donc toujours attendre la fin du processus 'plaques' pour s'exécuter.

Les deux sous-tâches 'montage' et 'disposer' sont synchronisées de la même façon.

L'enchaînement décrit ci-dessus pourra être réalisé par le fragment de code suivant :

```
gel_acryl :- ..., !, plaques(P), montage(P,M), disposer(M,D).

plaques(Y) :- true, !, ..., Y = fin_poser.
montage(fin_poser,Y) :- true, !, ..., Y = fin_tourner.
disposer(fin_tourner,Y) :- true, !, ..., Y = fin_poser.
```

On peut remarquer que, comme la tête de clause 'plaques' ne contient pas de variable instanciée, et que la garde est immédiatement satisfaite, l'exécution de la sous-tâche 'plaques' peut commencer sans condition préalable.

III.5.2. Répétition d'une même suite d'actions (boucle)

Trois sous-tâches ont été identifiées au premier niveau de décomposition. Chacune d'elles correspond à la répétition (1, 2 ou 3 fois) d'une même suite d'actions.

plaques : *répéter deux fois :*
saisir une plaque dans le distributeur de plaques
poser la plaque au site de montage.

montage : *répéter trois fois :*
saisir une pince dans le distributeur de pinces
fixer la pince au site de montage
tourner l'assemblage de 90 °

disposer : *répéter une fois :*
saisir l'assemblage au site de montage
poser l'assemblage au site de dépôt.

Il serait dès lors intéressant de disposer d'une construction permettant de décrire l'enchaînement des actions d'une même suite, et de dire le nombre de fois que cette suite doit être exécutée. Cette construction de boucle va être effectuée pour la sous-tâche 'plaques'. Le principe en est le même pour les deux autres sous-tâches 'montage' et 'disposer'. Il ne sera dès lors pas décrit pour celles-ci.

Comme nous l'avons vu à la section III.1., la primitive saisir a deux arguments :
Objet1 : l'objet qui doit être saisi,
Site1 : le site où se trouve l'objet à saisir.

Il en va de même pour la primitive poser, dont les deux arguments sont :
Objet2 : l'objet qui doit être posé,
Site2 : le site où l'objet doit être posé.
Les valeurs des arguments Objet1 et Objet2 pour l'exécution de la tâche 'plaques' sont identiques car l'objet à saisir est le même que l'objet qui doit être posé; Ces deux arguments auront la valeur 'plaque', l'argument Site1 aura la valeur 'distrib_plaques' et Site2, 'site_montage'.

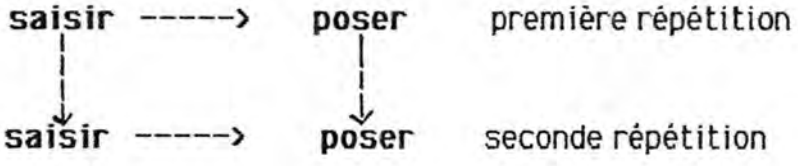
Les deux actions qui doivent être répétées (saisir puis poser) peuvent être décrites comme étant un empilement. L'exécution de la tâche 'plaques' nécessite donc la répétition de deux empilements.

La tâche 'plaques', comme nous l'avons vu à la section précédente, doit être synchronisée avec la tâche 'montage'. Elle aura donc un argument correspondant à la variable partagée de synchronisation.
La procédure empiler sera chargée d'exécuter une série d'empilements. Cinq arguments seront à cette fin nécessaires : le nombre d'empilements à exécuter, l'objet, le site 1, le site 2, une variable de synchronisation qui sera instanciée en fin d'exécution des deux empilements. Cette procédure sera appellée dans le corps de la clause de la tâche plaques, de façon à ce que les deux empilements décrivant cette tâche soient exécutés.
La clause de la tâche plaques sera donc la suivante :

```
plaques(Y) :- true, !,
        empiler(2,plaque,distrib_plaques,site_montage,Y).
```

Voyons maintenant comment cette procédure *empiler* peut être définie.

Comme on l'a vu, les contraintes de séquence entre les actions primitives de ces deux empilements peuvent être schématisées de la façon suivante :



On peut considérer chacune des deux répétitions de saisir comme étant exactement la même action puisque le distributeur de plaques (tout comme le distributeur de pinces) est conçu de façon à présenter l'objet à prendre toujours de la même manière et au même endroit.

Les deux répétitions de poser ne sont pas tout à fait identiques car la description du site de pose est légèrement modifiée pour la seconde plaque. En effet,

l'épaisseur de la première plaque qui est déposée fait que la seconde plaque doit être placée ± 1/2 cm plus haut que le site original. Une solution à cette modification du site serait de décrire la primitive 'poser' en prescrivant une approche progressive en fonction des coordonnées du site décrit.

Exemple : *poser au site 1*

--> à partir de 2 cm au-dessus du site 1, on répète la séquence :
- approcher de 3 mm
- tester si on touche la plaque

Quand on touche le site 1, on lâche la plaque, ou, de façon plus générale, l'objet que l'on tenait.

Si le mécanisme d'interrupteur chargé de détecter le toucher en établissant un contact tombe en panne, il ne faut pas approcher indéfiniment et risquer de défoncer le support. Un nombre maximum de répétitions de la séquence approcher-tester serait dès lors déterminé. Si ce nombre est atteint sans que la plaque n'ait été déposée, ce serait l'échec de la primitive 'poser'.

Cette extension de la primitive 'poser' n'a pas été considérée ici, on suppose donc que la différence du site de dépôt entre la première et la seconde plaque est négligeable, et que les deux répétitions de l'action de poser sont exactement identiques.

Afin d'obtenir les mêmes contraintes pour saisir et poser dans les deux répétitions, deux contraintes fictives vont être ajoutées, qui signaleront simplement que la répétition peut commencer :

Ces deux contraintes supplémentaires permettent de généraliser la description d'une répétition, de façon à ne pas faire de la première répétition un cas particulier.

On peut donc dire que, pour toute répétition, on peut exécuter saisir si le saisir de la répétition précédente est terminé, et que poser peut être exécuté quand le poser de la répétition précédente et le saisir de la même répétition sont terminés. Le principe appliqué pour implémenter ces contraintes est le principe de synchronisation de processus décrit à la section III.5.1.

La synchronisation nécessaire entre les tâches saisir et poser des répétitions doit être décrite au niveau du code des deux procédures saisir et poser.

La primitive exécutant la *tâche saisir* devra comporter cinq arguments : l'objet à saisir, le site de saisie, l'appareil physique qui sera utilisé (cfr section III.3.1., point c), deux variables de synchronisation S_in et S_out.

La première variable S_in est partagée en consommation avec le saisir de la répétition précédente, qui est chargé de l'instancier. Elle aura donc, dans la tête de la clause de saisir, la valeur fin_saisir, de façon à ce que le saisir soit suspendu tant que le saisir de la répétition précédente n'est pas terminé (principe de la section III.5.1.).

La seconde variable S_out est partagée en production avec la répétition suivante. Elle sera instanciée à la fin de saisir et permettra ainsi au saisir de la répétition suivante de commencer.

La primitive saisir est donc partiellement définie de la façon suivante :

saisir(Objet,Site,R,fin_saisir,S_out) :- true, !, ..., S_out = fin_saisir.

La primitive exécutant la *tâche poser* devra, quant à elle, comporter six arguments : l'objet à poser, le site de pose, l'appareil physique imposé (cfr section III.3.1., point c), trois variables de synchronisation P_in, S_out et P_out.

La première variable P_in est partagée en consommation avec le poser de la répétition précédente qui devra l'instancier. Elle correspondra, dans la tête de la clause poser, à la valeur 'fin_poser'. Le poser d'une répétition sera dès lors suspendu tant que celui de la répétition précédente ne sera pas terminé.

La deuxième variable S_out est partagée en consommation avec le saisir de la même répétition. Elle sera instanciée à la fin de l'exécution de saisir à la valeur fin_saisir. C'est cette valeur qui sera reprise dans la tête de la clause de poser, de façon à ce que l'action de poser ne puisse débuter avant la fin de l'action saisir de la même répétition. Il faudra donc que ces deux premières variables de synchronisation soient instanciées avant que poser ne puisse commencer.

La troisième variable P_out est partagée en production avec le poser de la répétition suivante. Elle sera instanciée à fin_poser à la fin de l'exécution de l'action de poser de façon à ce que le poser de la répétition suivante puisse débuter.

Le fragment de code suivant correspond à la primitive poser.

poser(Objet,Site,R,fin_saisir,fin_poser,P_out) :- true, !, ...,
P_out = fin_poser.

Nous pouvons, à partir de cette synchronisation des actions saisir et poser des deux répétitions, déduire la nécessité pour toute répétition d'empilement d'être synchronisée avec la précédente. Une répétition doit dès lors partager deux

variables avec la précédente : la première servant à synchroniser (selon le principe de la section III.5.1.) les deux saisir et la seconde, à synchroniser les deux poser. Les valeurs auxquelles ces deux variables de synchronisation seront instanciées seront respectivement fin_saisir et fin_poser. Comme la première répétition peut débuter sans condition préalable, les deux variables de synchronisation correspondant aux contraintes de début de répétition seront instanciées dans le premier appel.

Une procédure plus élémentaire que celle empiler (soit e_empiler) sera décrite et aura pour arguments les arguments de empiler plus les deux variables de synchronisation dont la nécessité vient d'être montrée. La fonction remplie par cette procédure est la même que celle de la procédure empiler : exécuter N empilements d'un objet à partir du site1 vers le site2 et signaler la fin de l'exécution de ces N empilements en instanciant la variable de synchronisation de niveau supérieur (=argument de plaques).

La clause correspondant à la procédure empiler peut être définie de la façon suivante :

```
empiler(N,Objet,Site1,Site2,Y) :- true, !,
    e_empiler(N,Objet,Site1,Site2,fin_saisir,fin_poser,Y).
```

Nous allons, pour la construction de la procédure *e_empiler*, définir une situation générale et une condition d'arrêt.

◇ La situation générale est celle où les N répétitions n'ont pas encore été effectuées. L'argument correspondant au nombre de répétitions restant à effectuer est dans ce cas supérieur à 0. Cette situation est identifiée dans la partie passive de la clause lui correspondant par un test vérifiant la positivité de N.

Si cette situation se présente, il faut exécuter une action de saisir en fonction des variables de synchronisation, une action de poser qui respecte aussi la synchronisation, et un appel récursif à e_empiler pour éventuellement effectuer la répétition suivante. Les arguments de cet appel récursif sont : le nombre de répétitions restant à effectuer diminué de un, l'objet et les sites inchangés, les variables de synchronisation de répétition qui sont instanciées en fin de saisir et poser, ainsi que la variable de synchronisation nécessaire pour la tâche plaques (instanciée seulement lorsque toutes les répétitions auront été totalement exécutées).

◇ La condition d'arrêt est vérifiée lorsque toutes les actions correspondant aux répétitions successives ont été considérées comme objectifs à résoudre (N est alors égal à 0), et lorsque toutes ces actions ont été exécutées (les variables de synchronisation des actions primitives terminales sont alors instanciées). Cette situation est identifiée lors de l'évaluation de la clause lui correspondant grâce à l'unification de cette tête de clause avec l'appel e_empiler. En effet, les arguments

de la tête de clause correspondant au nombre de répétitions restant à effectuer et aux deux variables de synchronisation de répétition sont respectivement instanciés aux valeurs 0, fin_saisir et fin_poser. L'unification de l'appel et de la tête de clause sera donc suspendue tant que la condition d'arrêt ne sera pas vérifiée au niveau de l'appel.

Lorsque cette situation d'arrêt est atteinte, la seule chose à faire est d'instancier dans le corps de la clause la variable signalant la fin d'exécution de toutes les répétitions.

◊ Les parties passives des deux clauses de la procédure e_empiler permettent donc de différencier sans aucune ambiguïté (et ce, même si les évaluations sont réalisées en parallèle-OU) la situation générale et la vérification de la condition d'arrêt.

Le code de la procédure e_empiler correspond aux deux clauses suivantes :

```
e_empiler(N,Objet,Site1,Site2,S_in,P_in,Y) :-
        N > 0, M is N - 1, !,
        saisir(Objet,Site1,Rob,S_in,S_out),
        poser(Objet,Site2,Rob,P_in,S_out,P_out),
        e_empiler(M,Objet,Site1,Site2,S_out,P_out,Y).
e_empiler(0,Objet,S1,S2,fin_saisir,fin_poser,Y) :-
        true, !, Y = fin_empiler.
```

Nous pouvons remarquer que, grâce au fait que l'appel récursif à la procédure e_empiler dans le cas de la situation générale peut être unifié directement (aucune suspension ne peut se produire, puisque les arguments de la tête de clause de la situation générale sont des variables non instanciées), la réduction de cet appel permet de considérer au plus tôt comme objectifs à résoudre les actions de saisir et poser de toutes les répétitions.

Afin d'illustrer cette remarque, une simulation de la résolution du prédicat **e_empiler(2,...,fin_saisir,fin_poser,Y)** est proposée ci-dessous.

Cette simulation est essentiellement basée sur l'évolution du contenu de la file d'ordonnancement au cours des différentes étapes de résolution.

Succession des états de la file d'ordonnancement :

1) e_empiler(...,fin_saisir,fin_poser,S1_out,Y)

réduction de e_empiler

2) saisir(...,fin_saisir,S1_out)
   poser(...,fin_poser,S1_out,P1_out)
   e_empiler(...,S1_out,P1_out,Y)

réduction de e_empiler

3) saisir(...,fin_saisir,S1_out)
   poser(...,fin_poser,S1_out,P1_out)
   saisir(...,S1_out,S2_out)
   poser(...,P1_out,S2_out,P2_out)
   e_empiler(0,...,S2_out,P2_out,Y)

exécution de saisir --> S1_out := fin_saisir

4) poser(..., *fin_saisir*,P1_out)
   saisir(..., *fin_saisir*,S2_out)
   poser(...,P1_out,S2_out,P2_out)
   e_empiler(0,...,S2_out,P2_out,Y)

suspension de poser

5) saisir(...,fin_saisir,S2_out)
   poser(...,P1_out,S2_out,P2_out)
   e_empiler(0,...,S2_out,P2_out,Y)
   poser(...,fin_saisir,P1_out)

exécution de saisir --> S2_out := fin_saisir (seulement possible si un second appareil est disponible)

6) poser(...,P1_out, *fin_saisir*,P2_out)
   e_empiler(0,..., *fin_saisir*,P2_out,Y)
   poser(...,fin_saisir,P1_out)

suspensions de poser et e_empiler

7) poser(...,fin_saisir,P1_out)
   poser(...,P1_out,fin_saisir,P2_out)
   e_empiler(0,...,fin_saisir,P2_out,Y)

exécution de poser --> P1_out := fin_poser

8) poser(..., *fin_poser*,fin_saisir,P2_out)
   e_empiler(0,...,fin_saisir,P2_out,Y)

exécution de poser --> P2_out := fin_poser

9) e_empiler(0,...,fin_saisir, *fin_poser*,Y)

exécution de e_empiler --> Y := fin_empiler

10) []

Les principales procédures du programme GHC réalisant l'application des gels acryliques ont été décrites. Le lecteur trouvera en annexe C le code complet de l'application. Par l'étude des qualités et des défauts du code obtenu, nous allons évaluer l'adéquation de GHC à la robotique.

III.6. Adéquation de GHC à la robotique

Une discussion visant à vérifier si le langage logique parallèle GHC est approprié pour la rédaction d'applications robotiques va maintenant être proposée. Nous allons pour cela analyser les différents avantages et inconvénients issus de l'utilisation de GHC dans ce domaine.

Le premier avantage n'est pas une exclusivité de GHC, c'est le parallélisme. Ce parallélisme est très intéressant au niveau de la description de l'enchaînement des tâches : il suffit de déterminer les contraintes de séquencement, toutes les autres tâches seront automatiquement effectuées en parallèle.

Les seules contraintes envisagées ici sont les contraintes de séquencement car nous nous plaçons dans le contexte limité des applications robotiques. Dans ce domaine, il est en effet peu vraisemblable d'avoir à considérer des contraintes d'autres types. Une contrainte d'exclusion mutuelle entre deux tâches (les deux tâches ne peuvent pas être exécutées en même temps) n'apparaîtra certainement pas souvent dans la description d'une application robotique.

Il faut toutefois remarquer qu'au niveau du temps d'exécution, le parallélisme n'améliore les performances que si plusieurs appareils sont disponibles pour l'exécution de l'application. Ce parallélisme intervient non seulement au niveau de l'exécution des actions, mais aussi au niveau de la sélection de l'appareil qui sera utilisé en fonction des priorités qui sont octroyées.

Le mécanisme utilisé pour synchroniser les processus en exploitant les règles de suspension a l'avantage de permettre de définir des boucles (qui correspondent à ce qui a été défini comme des répétitions d'une même suite d'actions, à la section III.5.2.). Ces boucles sont "aplaties" à l'exécution, c'est-à-dire que toutes les actions de chacune des itérations sont directement ordonnancées comme objectifs restants à résoudre. Les différentes itérations peuvent donc toutes être effectuées en parallèle, pour autant qu'elles satisfassent aux règles de suspension. Cela revient à dire que chaque action sera exécutée au plus tôt et ce, indépendamment de

l'itération de laquelle elle est originaire. Cette utilisation de la synchronisation est primordiale pour la description d'applications robotiques où il est fréquent de trouver des séquences d'actions qui se répètent un certain nombre de fois.

L'énoncé de ces deux avantages pourrait désigner GHC comme un langage bien approprié à la rédaction d'applications robotiques. Nous devons cependant aussi considérer un inconvénient de GHC qui, à lui seul, remet en question l'adéquation de GHC pour de telles applications.

Le principal inconvénient de l'utilisation de GHC dans le cadre d'applications robotiques est le manque de lisibilité du code résultant au niveau de la synchronisation des tâches. En effet, bien que les différents niveaux de description des tâches définis par l'utilisateur apparaissent clairement dans le code, la façon dont les contraintes de synchronisation entre les tâches de chaque niveau sont traduites n'est pas d'une grande clarté. Cet aspect de synchronisation des tâches étant primordial pour la description de toute application robotique, il est regrettable que le mécanisme de synchronisation prescrit par l'utilisation de GHC ne soit pas plus simple. Ce manque de simplicité complique la rédaction d'applications robotiques en GHC et accroît le risque de conception de programmes incorrects.

C'est une des raisons pour lesquelles une description systématique des synchronisations entre tâches fait partie de la méthodologie de conception d'applications robotiques proposée au chapitre suivant. La vérification de la validité des synchronisations y est assurée, de façon à garantir l'obtention d'un programme GHC correct.

Chapitre IV

Méthodologie de conception d'applications robotiques en GHC

IV.1. Notion de programme correct

IV.1.1. Introduction

Quel que soit le langage de programmation utilisé, et quelles que soient les applications visées, il est primordial que les programmes écrits soient corrects (réalisent leurs spécifications et se terminent). Il faut donc disposer de méthodes permettant de vérifier la validité de programmes. Ces méthodes ne sont pas destinées à détecter la présence d'erreurs (comme le font les tests de programmes), mais bien à prouver l'absence d'erreurs dans le programme.

Deux possibilités existent : démontrer la validité d'un programme qui existe déjà ou construire un programme correct.

La *première solution* nécessite tout d'abord de comprendre la façon dont le programme a été écrit, avant d'effectuer la démonstration. Le programme peut être correct tout en ayant un code à la limite de la lisibilité, ce qui rend la preuve plus compliquée à effectuer. De plus, si la preuve ne réussit pas, on ne peut en déduire que le programme est faux.

La *seconde solution* est de loin la meilleure car en une seule étape sont effectuées la rédaction du programme et la démonstration de sa validité. Cette seconde approche est donc celle qui sera utilisée ici.

Nous nous baserons principalement sur les résultats obtenus par Hogger (cfr [Ho84]) dans le cadre des programmes logiques.

IV.1.2. Définition de "programme logique correct"

La validité de tout programme est définie en fonction de la spécification du problème auquel le programme apporte une solution. La spécification d'un

programme logique peut être définie comme étant une assertion décrivant la relation liant les différents paramètres du programme.

Dans le cadre de la programmation logique, la vérification d'un programme nécessite de déterminer si les solutions obtenues sont correctes (vérifient la spécification du programme), et si toutes les solutions correctes pour le problème posé peuvent être obtenues à partir du programme.

La *validité partielle* d'un programme logique est définie comme étant la propriété selon laquelle toute solution obtenue à partir du programme est correcte par rapport aux spécifications. Un programme logique est donc partiellement correct s'il ne génère aucune solution qui violerait les spécifications (si toute solution générée est une solution spécifiée).

La *complétude* est la propriété selon laquelle toute solution correcte, pour un objectif donné à résoudre, peut être obtenue à partir du programme. Un programme logique n'est pas complet quand il ne contient pas assez d'informations à propos de la relation citée dans l'objectif pour générer toutes les solutions correctes existantes.

Un *programme logique totalement correct* vérifie ces deux propriétés, c'est-à-dire que les solutions qu'il génère sont exactement celles déterminées par les spécifications.

IV.1.3. Définition d' "algorithme logique correct"

Un algorithme logique est le résultat de l'utilisation d'une stratégie de contrôle pour l'exécution d'un programme logique.

Il est important de considérer la stratégie de contrôle en plus du programme lui-même car si la stratégie n'est pas équitable, des solutions qui pourtant devraient logiquement être possibles à obtenir ne seront jamais produites.

Une stratégie en profondeur d'abord, telle qu'elle est utilisée pour le Prolog séquentiel, n'est pas équitable car elle a pour but de développer le sous-objectif courant jusqu'à sa conclusion avant de développer les autres sous-objectifs.

Par contre, une stratégie en largeur d'abord, telle qu'elle est utilisée pour l'interpréteur de GHC (cfr chapitre II), développe l'ensemble des sous-objectifs en considérant une étape de résolution à la fois pour chacun à tour de rôle. Ils sont donc résolus de façon équitable, ce qui permet de produire toutes les solutions finies (déductibles par un nombre fini de réductions) dans un délai fini. Il y a toutefois une exception à cette équité : lorsque le nombre de sous-objectifs à considérer est infini, il est possible qu'un sous-objectif partiellement développé ne soit jamais entièrement résolu (l'interpréteur, considérant l'infinité des autres

sous-objectifs, ne reviendra pas à ce sous-objectif pour lui appliquer l'étape suivante de la résolution). Ce cas se présente parfois pour des procédures prédéfinies.

Cette distinction entre le programme et la stratégie de contrôle peut être, tout comme suggéré par Kowalski dans [Ko79a] et [Ko79b], représentée par la formule "Algorithm = Logic + Control" . La composante logique définit la partie de l'algorithme spécifique au problème, c'est-à-dire le programme, alors que la composante de contrôle concerne le comportement de l'algorithme, sans affecter sa signification.

Un algorithme logique est l'exécution d'un programme logique selon une stratégie de contrôle. Un algorithme logique *se termine* si et seulement s'il comporte un nombre fini d'exécutions qui sont toutes de longueur finie.

Un *algorithme logique totalement correct* produit, en un temps fini, exactement toutes les solutions désignées par ses spécifications.

IV.1.4. Vérification d'un programme logique

Les critères suffisants devant être vérifiés par un programme logique correct sont au nombre de trois :

(1) le programme réalise sa spécification (validité partielle, cfr section IV.1.2.),

(2) le programme est complet en fonction de sa spécification (complétude, cfr section IV.1.2.),

(3) toutes les exécutions du programme (en nombre fini) sont de longueur finie (terminaison d'algorithme logique, cfr section IV.1.3.).

La section suivante décrit une proposition de méthodologie de conception d'une application robotique dans le langage logique parallèle GHC.

IV.2. Méthodologie de conception d'une application robotique en GHC

IV.2.1. Contexte d'applicabilité de la méthodologie

Le langage dans lequel les applications sont rédigées est le langage GHC. Les applications sont des applications décrivant des tâches à effectuer par un certain nombre d'appareils : ce sont des applications robotiques. Cette méthodologie est avant tout destinée à des utilisateurs non spécialistes, pour leur permettre de créer les applications dont ils donnent la description de l'enchaînement des tâches. La partie des applications qui est statique pour toute description de tâches (gestion des appareils et lien avec la couche objet) ne sera donc pas abordée. Elle peut en

effet être décrite une seule fois pour toutes les applications destinées au même environnement de travail. Tel n'est pas le cas de la description des tâches à effectuer.

La partie concernant la description et l'enchaînement des tâches peut être systématiquement transformée pour passer d'une exécution séquentielle à une exécution parallèle. En effet, la méthodologie proposée à la section IV.2.4. ci-dessous permet de transformer la description des tâches d'une application en Prolog séquentiel en une description en GHC. Le passage du séquentiel au parallèle sera souvent justifié par l'augmentation du parc d'appareils disponibles. Rappellons que la méthodologie décrite ici n'aborde pas la partie de l'application décrivant les appareils car la description du nouveau parc ne peut être directement déduite à partir de celle de l'ancien.

Il est important de préciser qu'il ne s'agit donc pas d'une dérivation de programme à partir de spécifications (synthèse de programme), mais d'une dérivation de programme à partir de fragments de code Prolog (transformation de programme). En effet, le but est de partir d'une description d'une application séquentielle en Prolog et d'y ajouter les renseignements nécessaires pour en faire une description parallèle en GHC. C'est pourquoi la description de l'enchaînement des tâches est rédigée en Prolog séquentiel : cet enchaînement est considéré comme une partie d'une description complète de l'application en Prolog séquentiel.

IV.2.2. Caractéristiques des descriptions de tâches d'une application

Les descriptions de tâches de l'application seront séparées en trois parties distinctes : une première partie reprend la description de l'enchaînement séquentiel des tâches à effectuer, une deuxième décrit les tâches qui peuvent être effectuées en parallèle, et une troisième décrit chacune des tâches en fonction des primitives disponibles ou d'un enchaînement de sous-tâches.

a) enchaînement séquentiel des tâches

Cet enchaînement séquentiel correspond à la description des tâches telles qu'elles seraient exécutées en l'absence de tout parallélisme. Cette absence de parallélisme pourrait par exemple être dûe à la présence d'un appareil unique. Dans ce cas, toutes les tâches doivent être exécutées par le même appareil, et leur séquencement est obligatoire. Cette description correspond donc exactement à celle donnée dans le mémoire de Claude Vanhemelryck (aucun parallélisme n'est envisagé), cfr [Va85].

Le langage utilisé pour décrire cet enchaînement de tâches est le Prolog séquentiel pur. On exclut l'utilisation du "cut". En effet, dans le cadre d'applications robotiques, l'exécution d'une tâche correspond à un prédicat qui ne

peut être résolu que d'une seule manière. Aucun rétroparcours dans une conjonction d'objectifs correspondant à de tels prédicats et réalisant un enchaînement séquentiel de tâches n'entraînerait une seconde résolution d'un tel prédicat. L'utilisation du "cut" n'est donc pas justifiée pour cette description en Prolog de l'enchaînement séquentiel de tâches.

La stratégie liée à l'exploitation d'un programme écrit en Prolog évalue les sous-objectifs de façon séquentielle de la gauche vers la droite. Les tâches devant être exécutées en premier lieu correspondent donc à des objectifs situés à gauche de la liste d'objectifs à résoudre.

Exemple : • *enchaînement séquentiel*: **1)** sous_tâche_1
2) sous_tâche_2
3) sous_tâche_3

• *description en Prolog*:

tâche :- sous_tâche_1, sous_tâche_2, sous_tâche_3. **(1)**

Cet enchaînement séquentiel correspond à un fragment du code Prolog décrivant une application complète.

b) parallélisme entre tâches

Comme nous l'avons vu à la section IV.2.1., le principe de la transformation de programme présenté ici est de partir d'une application séquentielle décrite en Prolog. Une application parallèle en GHC est obtenue sur base de parties de code Prolog correspondant aux enchaînements de tâches (point a), et de renseignements concernant le parallélisme possible (point b). C'est la raison pour laquelle, bien qu'aux deux points a et b soient exprimées des relations de précédence entre tâches, ces deux parties de description de tâches sont explicitement séparées.

Le parallélisme entre tâches est décrit sous forme de *contraintes de succession imposées à certaines tâches*.

Plusieurs cas de succession de deux tâches (X et Y) peuvent être envisagés :
1) Le début de la tâche X doit précéder dans le temps le début de la tâche Y.
2) La fin de la tâche X doit précéder dans le temps la fin de la tâche Y.
3) La fin de la tâche X doit précéder dans le temps le début de la tâche Y.

Les contraintes de succession considérées ici correspondent au troisième cas. La contrainte **succession(X,Y)** sera représentée graphiquement par :

X ---> Y

et signifiera que la fin de la tâche X doit précéder dans le temps le début de la tâche Y (la tâche Y ne peut débuter avant la fin de la tâche X).

Hormis ces contraintes, les autres tâches peuvent donc toutes être exécutées en parallèle.

Pour déterminer ces contraintes, on suppose que le nombre d'appareils disponibles n'est pas limité : deux tâches ne seront théoriquement pas séquentialisées pour cause de manque d'appareils. Le parallélisme envisagé est donc un parallélisme maximal.

Exemple de contrainte pour la clause (1) ci-dessus :

sous_tâche_2 ---> sous_tâche_3 **(2)**

La sous-tâche 3 ne peut débuter qu'après la fin de la sous-tâche 2, la sous-tâche 1 peut donc être exécutée en parallèle avec la sous-tâche 2, ou avec la sous-tâche 3.

On aurait pu envisager de donner plutôt les *sous-ensembles de tâches qui peuvent être exécutées en parallèle*. La situation décrite par **(1)** et **(2)** ci-dessus aurait été exprimée sous la forme de deux clauses réunies par un opérateur 'OU' (à la place d'une seule contrainte pour exprimer la succession) :

parallèle(sous_tâche_1,sous_tâche_2)
parallèle(sous_tâche_1,sous_tâche_3)

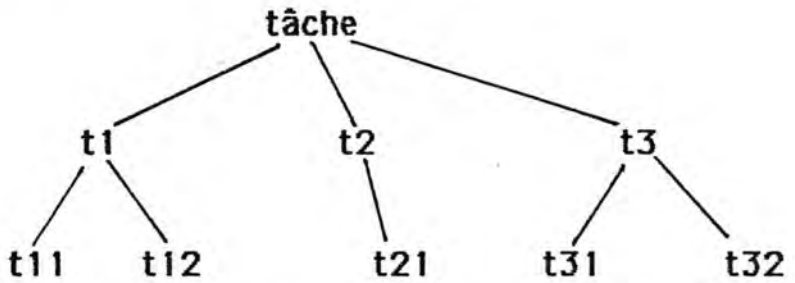
Il nous a semblé plus simple pour l'utilisateur de déterminer les contraintes de succession plutôt que les possibilités de parallélisme.
De plus, le nombre de contraintes de succession sera en général beaucoup moins élevé que le nombre de possibilités de parallélisme pour les actions. La longueur de cette partie des descriptions en sera d'autant plus réduite.
Un autre critère utilisé pour rejeter la description du parallélisme au moyen d'ensembles de sous-tâches parallèles est la possibilité qui existe de faire une vérification quant à la cohérence des contraintes fournies par l'utilisateur (voir point suivant). Il serait en effet beaucoup plus complexe d'effectuer une telle vérification sur base d'ensembles d'actions qui peuvent être effectuées en parallèle, que sur base de contraintes de succession.

c) description des tâches

En ce qui concerne la description des tâches, l'utilisateur peut travailler par raffinements successifs. Une décomposition directe en primitives pour une grosse application peut donc être évitée.

La description d'une application peut être schématisée sous forme d'un arbre.

Exemple :



Les tâches du dernier niveau (les feuilles de l'arbre) correspondent à des appels aux primitives offertes par la couche objet. Elles ne nécessitent pas de description supplémentaire.

Les descriptions des tâches des niveaux supérieurs seront caractérisées par trois parties :

(1) enchaînement des sous-tâches
(2) contraintes de succession des sous-tâches
(3) description des sous-tâches (description en trois parties si les sous-tâches ne sont pas primitives, identification de la primitive sinon).

Les descriptions de tâches telles qu'elles viennent d'être présentées permettent de décrire clairement l'enchaînement des tâches à effectuer. Elles sont supposées correctes et cohérentes. Voyons comment il est possible de vérifier leur validité.

IV.2.3. Vérification des descriptions de tâches

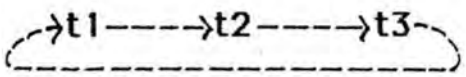
Les descriptions de tâches risquent de comporter trois types d'erreurs : une incohérence au niveau des contraintes de succession, une incohérence entre l'enchaînement séquentiel des tâches et les contraintes de succession, une erreur dans la description des tâches. Si au moins une des trois situations problématiques se présente, les descriptions de tâches ne pourront être utilisées pour la transformation en un programme GHC. Elles devront donc être corrigées. Il sera admis dans la suite que les descriptions utilisées pour l'obtention du programme GHC ne comportent ni incohérence, ni erreur.

a) incohérence au niveau des contraintes de succession

Une incohérence au niveau des contraintes de succession se présente lorsqu'une contrainte prescrit entre deux tâches un ordre de succession inverse de celui prescrit par l'ensemble des autres contraintes pour ces deux mêmes tâches.

Ce problème est caractérisé par la présence d'un circuit dans le graphe représentant les contraintes de succession.

exemple :



Selon cette description, t1 devrait être exécutée avant t3, et t3 avant t1.

La détection d'une incohérence au niveau des contraintes de succession nécessite l'application d'une technique de détection de circuit classique. Les contraintes de succession seront représentées sous forme d'un graphe. Chaque tâche correspond à un sommet du graphe et une contrainte entre deux tâches est signalée par un arc entre les deux sommets correspondants.

La détection d'un circuit dans un graphe est réalisée grâce à une décomposition du graphe en niveaux. Si cette décomposition n'est pas possible, c'est la preuve de l'existence d'un circuit. Il ne nous a pas semblé opportun de développer cette théorie dans le cadre de ce travail. Le lecteur consultera [Fi], [Beg70] ou [Ro69] pour de plus amples informations.

b) incohérence entre l'enchaînement séquentiel des tâches et les contraintes de succession

Une incohérence entre l'enchaînement séquentiel des tâches et les contraintes de succession peut être caractérisée par la situation où une contrainte de succession prescrit entre deux tâches un ordre de succession inverse de celui prescrit par l'enchaînement séquentiel pour les deux tâches concernées. Les deux tâches dans l'enchaînement séquentiel peuvent se succéder immédiatement, ou d'autres tâches peuvent les séparer. Quel que soit le cas, il est illogique de spécifier dans les contraintes de succession un ordre inverse de l'ordre de l'enchaînement séquentiel tel qu'il était utilisé pour l'application séquentielle en Prolog. En effet, la prise en considération d'une telle contrainte serait en contradiction avec l'application telle qu'elle avait été décrite en Prolog séquentiel.

Ce cas n'aurait pas dû être envisagé si on ne reprenait du Prolog séquentiel que la syntaxe, et non la sémantique. L'enchaînement serait alors considéré comme la description de la décomposition en sous-tâches, sans ordre d'exécution. Ce n'est toutefois pas ce qui est réalisé ici car l'enchaînemnet décrit en Prolog (soit **t :- t1, t2.**) correspond en fait à une contrainte de succession disant que **t1** précède **t2**. Les éventuelles incohérences doivent donc être détectées.

exemple :

enchaînement séquentiel : **t1, t2, t3**
contrainte : **t3 ---> t1**

L'enchaînement et la contrainte prescrivent que t1 devrait être exécutée avant t3, et t3 avant t1.

La détection d'une incohérence entre la description de l'enchaînement séquentiel des tâches et les contraintes de succession est réalisée par une comparaison de l'ordre des actions apparaissant dans les contraintes et de leur ordre dans l'enchaînement.

Pour chaque contrainte, il faut vérifier si les deux tâches considérées apparaissent dans le même ordre dans l'enchaînement. D'autres tâches peuvent se trouver entre les deux tâches concernées par la contrainte. La tâche située à gauche de l'opérateur ' ---> ' de la contrainte doit, dans l'enchaînement séquentiel, précéder (être à gauche de) la tâche qui est située à droite de la contrainte.

Exemples : (1) enchaînement : t1, t2, t3
contraintes : t1 ---> t2
t1 ---> t3
il n'y a pas d'incohérence par rapport à l'enchaînement

(2) enchaînement : t1, t2, t3, t4
contraintes : t3 ---> t4
t3 ---> t1
la première contrainte ne pose pas de problème mais la seconde présente une incohérence car elle spécifie t3,...,t1 alors qu'on trouve dans l'enchaînemnt t1,...,t3.

Si l'ordre présent dans l'enchaînement correspond à chaque contrainte, l'utilisateur n'a pas introduit d'incohérence dans sa description de l'application.

c) erreur dans la description des tâches

Deux types d'erreurs sont considérés dans le cadre de ce travail :

1. *Absence de description d'une tâche non primitive.*
Les tâches non primitives doivent être décrites de la manière présentée à la section IV.2.2. Si une tâche citée à un certain niveau de raffinement ne correspond à aucune description au niveau inférieur, l'erreur est signalée.

2. *Appel à une primitive inexistante.*

Les tâches primitives sont décrites comme des appels aux primitives offertes par la couche objet. Une erreur apparaît lorsqu'une primitive n'appartenant pas à l'ensemble des primitives disponibles est citée.

Ces deux types d'erreurs étant corrigés par l'utilisateur, on peut considérer que l'application sera construite à partir de descriptions de tâches dont la cohérence est assurée.

IV.2.4. Transformation de la description séquentielle des tâches en un programme GHC

a) propriétés de la transformation

Le programme GHC dérivé à partir des fragments de programme Prolog séquentiel devra vérifier certaines propriétés, de façon à pouvoir être considéré comme correct. Ces propriétés correspondent à celles qui ont été citées à la section IV.1.

La première propriété à envisager correspond en quelque sorte à la *validité partielle* (cfr section IV.1.2.). L'ordre dans lequel l'exécution des tâches prescrite par le programme GHC sera effectuée ne doit pas violer la description et l'enchaînement des tâches tels qu'ils apparaissent dans les descriptions de tâches en Prolog séquentiel. Les actions effectuées doivent correspondre exactement aux actions primitives de la description, et l'ordre dans lequel elles sont effectuées doit être cohérent par rapport à l'enchaînement séquentiel et aux contraintes de succession.

Conformément à la section IV.1.2., la deuxième propriété que le programme GHC devrait vérifier est la *complétude*. Le programme GHC devrait générer toutes les solutions correctes existantes. Dans le cadre des applications robotiques et du langage GHC, cette propriété ne semble pas être appropriée. En effet, la sémantique procédurale du langage GHC est telle qu'il n'y a pas de rétroparcours sur le choix de la clause utilisée pour résoudre un objectif (cfr section I.2.9.). Le programme GHC ne produira jamais qu'une seule solution, si une telle solution existe. La propriété de complétude ne sera donc pas appliquée dans ce cas-ci.

La troisième propriété est la *terminaison*. Elle est basée sur le programme et sur la stratégie de résolution utilisée pour l'exécution du programme. Nous avons vu à la section I.1.8. que des problèmes liés à l'exploitation du parallélisme pouvaient engendrer la non-terminaison de l'exécution d'un programme (interblocage ou blocage individuel permanent). Il faudra donc veiller à ce que de telles situations soient évitées.

Si la transformation d'un programme Prolog en un programme GHC préserve les première et la troisième propriétés, le programme transformé résultant sera considéré comme étant correct. Voyons maintenant comment le passage de la description séquentielle des tâches en Prolog au programme parallèle en GHC peut être effectué.

b) représentation des contraintes de succession

L'existence de contraintes de succession entre tâches implique la nécessité pour les processus correspondant à la résolution des tâches concernées de communiquer afin de pouvoir se synchroniser. En GHC, la communication entre processus n'est possible que si ceux-ci partagent des variables communes. Il faudra donc qu'une contrainte de succession entre deux tâches corresponde au partage d'une variable entre les deux processus correspondants. L'application de ce principe de synchronisation a déjà été illustrée dans le cadre de l'exemple de réservation aérienne (section I.5.3.) et dans le chapitre III décrivant l'application robotique (section III.5.1.).

Exemple : contrainte : **t1 ---> t2**
prédicats correspondant aux processus :
t1(V), t2(V).

Le processus correspondant à la tâche située à gauche de l'opérateur '--->' de la contrainte sera le producteur pour la variable partagée, tandis que le processus correspondant à la tâche de droite en sera un consommateur.

Il n'y aura pas nécessairement une variable par contrainte car une même tâche qui serait à l'origine de deux contraintes différentes n'entraînera la présence que d'une variable unique pour les deux contraintes. Plusieurs consommateurs seront donc possibles pour un même producteur.

Exemple : contraintes : **t3 <--- t1 ---> t2**
prédicats : **t1(V), t2(V), t3(V).**
Les deux contraintes signifient que la fin de la tâche t1 doit être signalée à la tâche t2 et à t3, avant que celles-ci ne puissent débuter. Il suffit que t1 soit producteur d'une valeur pour une variable qui serait partagée aussi par t2 et t3. Ces deux dernières tâches seraient les consommateurs de la variable. De cette façon, chaque tâche sera producteur au maximum d'une variable partagée.

Les tâches n'intervenant dans aucune contrainte ne comporteront pas de variable de synchronisation (ni à la production, ni à la consommation).

L'existence de variables partagées détermine le moyen de communication entre processus. Voyons maintenant comment cette communication doit être effectuée pour réaliser la synchronisation voulue.

La synchronisation sera réalisée en exploitant la règle de suspension (a) de GHC (cfr point c, section I.5.2.).

Le processus producteur pour la variable partagée instanciera cette variable à une certaine valeur lorsque la tâche correspondante aura été exécutée.

Le(s) processus consommateur(s) de la variable partagée spécifiera(ont) dans la(les) tête(s) de clause la valeur donnée par le producteur. Afin de respecter la règle de suspension (a), l'appel ne pourra être réduit que si son unification avec la tête de clause ne génère aucune exportation de liaison. Si le producteur n'a pas encore instancié la variable, l'unification de l'appel sera suspendue et la tâche correspondant au consommateur ne pourra être exécutée. Elle devra attendre la fin de la tâche productrice avant de commencer son exécution. Cela correspond bien à la sémantique liée aux contraintes de succession.

Exemple : contrainte : **t1 ---> t2**

clauses : **tâche :- ... t1(V), t2(V).** (1)
t1(V) :- ... V = fin. (2)
t2(fin) :-... (3)

La résolution des deux sous-objectifs spécifiés en (1) sera strictement séquentielle car t1 pourra être réduit directement, tandis que t2 devra attendre la terminaison de t1 (l'instanciation de la variable partagée V) avant que sa suspension ne soit levée et qu'il puisse à son tour être réduit.

La variable partagée sera instanciée par le processus producteur au niveau des primitives de la couche objet. On attend ainsi l'exécution effective de l'action avant de signaler la fin de la tâche. Il faut que la primitive qui instancie la variable soit celle qui est la dernière à être exécutée pour la tâche considérée, de façon à ce que l'instanciation corresponde bien à la fin de la tâche. Ce sera donc la primitive qui n'est origine d'aucune contrainte qui instanciera la variable. Deux cas peuvent se présenter.

Le premier correspond à la situation où il n'y a qu'une seule primitive (soit P) qui n'est origine d'aucune contrainte. Il n'y a dans ce cas aucun problème car cela signifie que, quel que soit l'ordre d'exécution des autres primitives, la primitive P sera toujours la dernière à être exécutée.

Le second cas est celui où plusieurs primitives ne sont origine d'aucune contrainte. Ce cas pose un problème, car on ne peut choisir arbitrairement une des primitives pour instancier la variable signalant la fin de la tâche principale. En effet, une instanciation effectuée par une seule primitive ne garantirait pas la fin d'exécution de la tâche car d'autres primitives étant exécutées en parallèle (et sans aucune

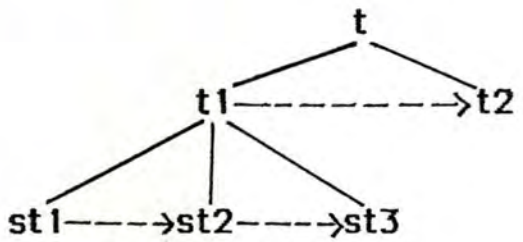
synchronisation imposée), celles-ci pourraient ne pas encore être terminées au moment de l'instanciation.

Des exemples illustrant ces deux cas sont donnés ci-dessous.

Premier cas : une seule primitive origine d'aucune contrainte

Exemple 1

- *arbre de description de la tâche*:



t:- t1(V), t2(V).

la tâche t1 est producteur de la variable V, et t2 en est consommateur.

- *description de la tâche t1 :*

1) *enchaînement en Prolog*: **t1 :- st1, st2, st3.**

Remarque : les trois sous-tâches de t1 sont des primitives.

2) *contraintes*: **st1 --> st2**
st2 --> st3

La sous-tâche st3 est donc la seule à n'être origine d'aucune contrainte. C'est à son niveau que sera instanciée la variable de synchronisation V car elle sera toujours la dernière sous-tâche à être exécutée pour t1.

3) *transformation en GHC*:

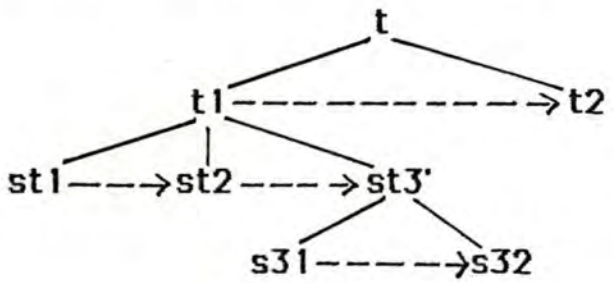
```
t1(V) :- true, !, st1(A), st2(A,B), st3(B,V).

st1(X) :- ... X = fin_1.
st2(fin_1,X) :- ... X = fin_2.
st3(fin_2,X) :- ... X = fin_3.
```

Exemple 2

Nous allons considérer la tâche décrite dans l'exemple 1, en modifiant la description de la sous-tâche st3 : celle -ci n'est plus une primitive. On va dès lors appliquer à cette sous-tâche la même règle en ce qui concerne l'endroit d'instanciation de la variable V.

- *arbre de description de la tâche*:



- *description de la sous-tâche st3'*

1) enchaînement en prolog

st3' :- s31, s32.

2) contrainte : **s31 ---> s32**

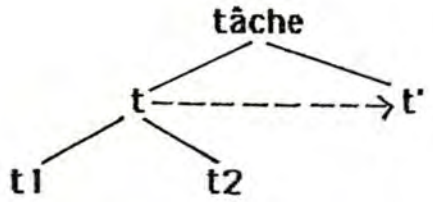
C'est au niveau de la tâche s32 (car s31 est origine d'une contrainte) que l'instanciation de la variable signalant la fin de st3' sera effectuée.

3) transformation en GHC

```
st3'(fin_2,V) :- true, !, s31(A), s32(A,V).
    s32(...,V) :- V = fin_3.
```

Second cas : plusieurs primitives origine d'aucune contrainte

- *arbre de description de la tâche*



tâche :- true, !, t(V), t'(V).

Dans cette description en GHC de tâche, t est producteur de V, et t' en est consommateur.

• *description en Prolog séquentiel de la tâche t*

t:- t1, t2.

Il n'y a pas de contrainte, aucune des deux sous-tâches t1 et t2 n'est origine d'une contrainte. Si on choisissait arbitrairement t2 pour instancier la variable de fin d'exécution de t et que t1 à ce moment n'était pas terminée, t' pourrait commencer avant que la tâche t ne soit entièrement exécutée.

Une façon de résoudre ce problème est d'ajouter une tâche fictive marqueur de fin de tâche et des contraintes artificielles pour revenir à la situation décrite dans le premier cas : il n'y aurait ainsi qu'une seule sous-tâche qui ne soit origine d'aucune contrainte . La tâche fictive est une tâche dont le seul but est d'instancier une variable de synchronisation.

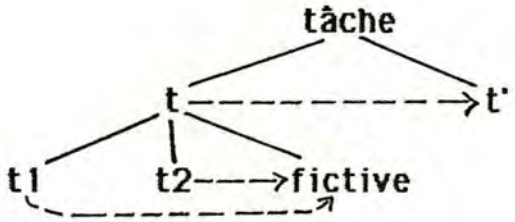
On ajoute une contrainte artificielle par sous-tâche qui n'est origine d'aucune contrainte. Une telle contrainte part de la tâche qui n'est pas origine et va vers la tâche fictive.

sous-tâche pas origine ---> tâche fictive

La tâche fictive ne peut ainsi commencer que lorsque toutes les sous-tâches seront terminées. C'est seulement à ce moment que la variable spécifiant la fin d'exécution de la tâche principale sera instanciée.

Exemple : modification par ajout de tâche fictive et de contraintes pour l'exemple donné ci-dessus.

• *arbre de description de la tâche :*



t :- t1, t2, fictive.

• *contraintes pour la description de t :*

t1 ---> fictive
t2 ---> fictive

La seule sous-tâche qui ne soit pas origine d'une contrainte étant la tâche 'fictive', c'est à son niveau que sera instanciée la variable de synchronisation. Le code GHC ainsi généré correspond à ce qui suit :

```
t(V) :- true, !, t1(V1), t2(V2), fictive(V1,V2,V).

    t1(V1) :- ... V1 = fin_1.
    t2(V2) :- ... V2 = fin_2.
    fictive(fin_1,fin_2,V) :- true, !, V = fin.
```

c) description des tâches

Si la tâche est primitive, sa description sera un simple appel à la primitive objet correspondante. Si elle n'est pas primitive, sa description sera un enchaînement de sous-tâches basé sur les contraintes et l'enchaînement définis dans les descriptions de tâches de l'application.

La *description en GHC d'une tâche primitive* comportera dans la tête de la clause correspondant à la tâche les éventuels arguments nécessaires pour la synchronisation (en fonction de ce qui a été présenté au point précédent). La garde de la clause sera vide car une seule clause correspondra à chaque tâche à décrire. Aucun test de sélection d'une clause candidate parmi l'ensemble des clauses d'une procédure ne devra être effectué. La clause sera candidate dès que l'unification de sa tête avec l'appel aura abouti. L'appel à la primitive sera effectué dans le corps de la clause. Une telle tâche sera donc décrite de la façon suivante :

```
tâche(...) :- true, !, primitive(...).
```

La *description en GHC d'une tâche non primitive* sera similaire à celle d'une tâche primitive, si ce n'est que l'appel à la primitive sera remplacé par une conjonction d'appels décrivant l'enchaînement des sous-tâches. Des variables seront ajoutées aux arguments des prédicats des sous-tâches si une synchronisation entre celles-ci est nécessaire. Les clauses décrivant de telles tâches auront dès lors la forme suivante :

```
tâche(...) :- true, !, ss_tâche_1(...), ss_tâche_2(...), ...
```

IV.2.5. Vérification du programme GHC

Le programme GHC ainsi obtenu par transformation est correct s'il est partiellement correct et s'il se termine (cfr point a de la section IV.2.4.).

a) validité partielle

L'exécution des tâches prescrite par le programme GHC ne doit violer en rien les descriptions de tâches de l'application. Deux points essentiels doivent dès lors être vérifiés : les primitives invoquées doivent être celles citées dans les descriptions de tâches et l'enchaînement des tâches doit vérifier les contraintes de succession et l'enchaînement séquentiel prescrits.

1) Cette correspondance entre les primitives invoquées dans le programme GHC et celles citées dans les descriptions de tâches est assurée par l'absence de modification des appels de primitives entre le code Prolog et le code GHC. Pour ce qui est des primitives invoquées, les actions générées correspondent donc bien aux descriptions de tâches.

Il faut cependant traiter séparément le cas où une tâche fictive est ajoutée pour avoir une sous-tâche terminale dans l'exécution d'une tâche. Cette tâche fictive ne modifie en rien la sémantique de l'application car elle ne fait appel à aucune primitive de la couche objet, elle instancie simplement une variable de synchronisation. L'ajout d'une telle tâche ne modifie pas les primitives invoquées, celles-ci seront donc bien celles qui ont été citées dans les descriptions de tâches.

2) L'enchaînement des tâches doit vérifier les contraintes de succession et l'enchaînement séquentiel prescrits. Il faut pour cela que les tâches soient effectuées en séquence quand il existe des contraintes de succession, et en parallèle si elles n'interviennent dans aucune contrainte.

La façon dont les contraintes de succession sont traduites en GHC (cfr section IV.2.4., point b) garantit que chaque couple de tâches intervenant dans une contrainte sera exécuté séquentiellement, la seconde tâche ne pouvant débuter qu'à la fin de la première. La transformation systématique par gestion des variables de synchronisation garantit que tous les cas de séquencement obligatoire seront traités correctement.

Tout comme expliqué dans [Ko74], la possibilité d'une exécution parallèle indépendante est offerte lorsque les différents sous-objectifs considérés dans le même appel n'ont pas de variables communes. Cela correspond à la façon dont le programme GHC est construit : les tâches qui n'interviennent pas dans la même contrainte de succession, et qui ne partagent donc pas de variable sont celles qui peuvent être exécutées en parallèle. Les possibilités de parallélisme sont donc maximales eu égard aux contraintes de précédence prescrites.

Le cas de l'ajout d'une tâche fictive ne modifie pas l'enchaînement des tâches tel qu'il a été défini dans les descriptions de tâches. La présence de la tâche fictive va de pair avec des contraintes artificielles entre cette tâche et chacune des tâches qui ne sont origine d'aucune des contraintes de base. Aucune contrainte artificielle n'est décrite entre deux sous-tâches, les relations entre les sous-tâches elles-mêmes ne sont pas modifiées au niveau des possibilités de parallélisme, la

modification n'intervient que dans les relations entre ces sous-tâches et la tâche fictive. On pourrait dire que l'enchaînement des sous-tâches est modifié car une tâche fictive a été ajoutée et se synchronise avec d'autres sous-tâches déjà définies. Il n'en est cependant rien car il faut considérer l'enchaînement des actions réellement effectuées par l'appareil. Cet enchaînement n'est pas modifié car la tâche fictive n'a aucune répercussion sur les actions exécutées, elle est simplement un moyen de synchronisation de tâches de plus haut niveau.

Nous avons ainsi argumenté que le programme GHC ainsi obtenu est partiellement correct.

b) terminaison

Pour que l'exécution du programme se termine, trois points doivent être vérifiés :
1) il faut qu'on n'introduise pas de nouvelles possibilités de boucles infinies,
2) il faut que les schémas de synchronisation producteur/consommateur n'introduisent pas de possibilités d'interblocage (cfr section I.1.8.),
3) il ne faut pas de blocage permanent. Un tel blocage peut avoir deux origines différentes :

1. un manque d'équitabilité de la stratégie de résolution employée
2. une incohérence dans le code du programme ou des coalitions contre un processus particulier (cfr section I.1.8.).

Ces trois points vont être considérés un à un, de façon à voir si des risques de non-terminaison du programme existent.

1) En ce qui concerne l'introduction de nouvelles *boucles infinies*, cette possibilité ne se présentera pas car la manière dont les descriptions de tâches ont été transformées en GHC ne comporte aucune modification de la description-même des tâches. C'est en effet principalement au niveau de la synchronisation que la transformation se situe. De nouvelles boucles infinies issues de la résolution des objectifs correspondant aux tâches à exécuter ne devraient donc pas se présenter.

2) Des *risques d'interblocage* auraient été introduits dans le cas où un premier processus (soit P1) serait producteur d'une valeur pour un second processus (soit P2) et où P2 serait producteur d'une valeur pour une variable consommée par P1. Une telle situation correspondrait à une synchronisation prescrivant que P1 doit précéder P2, et que P2 doit précéder P1. Or, nous avons vu à la section IV.2.3. que si une telle incohérence au niveau des contraintes de succession était détectée, la transformation ne serait pas effectuée. Ce problème d'interblocage ne se présentera donc pas pour le programme GHC.

3) Le *blocage permanent* pourrait être dû à l'emploi d'une stratégie de résolution non équitable. Nous avons vu au chapitre II (section II.5.3., point a) que la stratégie employée est la stratégie en largeur d'abord. Cette stratégie est équitable pour un ensemble de sous-objectifs car ceux-ci seront résolus au même rythme (chacun se voyant appliquer un pas de résolution à tour de rôle). Aucun problème d'inéquité ne peut donc résulter de l'emploi de la stratégie en largeur d'abord pour l'interpréteur du langage GHC. Le problème pouvant survenir dans le cas d'un nombre infini de sous-objectifs à résoudre (cfr section IV.1.3.) ne sera pas introduit par la tranformation en GHC. C'est en effet principalement au niveau des prédicats prédéfinis que ce problème risque de se produire, or, les procédures correspondant à ces prédicats ne sont pas modifiées.

Une seconde cause de blocage permanent serait la présence d'incohérences dans le programme. De telles incohérences pourraient avoir pour origine une tâche qui ne serait pas décrite. L'appel correspondant à cette tâche ne serait jamais résolu car aucune des procédures du programme ne pourrait être utilisée à cet effet. Ce problème ne se présentera pas car les descriptions de tâches sont supposées correctes et leur traduction est systématique. A toute tâche correspondra toujours une description.

Un problème de synchronisation pourrait aussi être à l'origine d'un blocage permanent. Un tel problème pourrait survenir dans trois situations :

I) un processus consommateur auquel ne correspondrait aucun producteur,

II) un blocage des producteurs,

III) une coalition entre consommateurs.

Voyons si ces situations pourraient se présenter.

I) Comme la description des contraintes de succession est effectuée de façon à assurer pour toute contrainte la présence d'une instanciation de variable partagée ainsi que la présence de la valeur de cette variable dans une tête de clause, on peut affirmer qu'à tout consommateur correspond un producteur. Les consommateurs ne seront donc jamais retardés indéfiniment à cause de l'absence d'un producteur.

II) Les processus producteurs ne seront jamais bloqués, si ce n'est dans la situation où ils sont consommateurs pour une autre variable. Comme des cas d'interblocage ne peuvent se produire, ils seront tôt ou tard satisfaits dans leur rôle de consommateur et pourront alors remplir leur second rôle de producteur.

III) Aucune coalition n'est possible entre les consommateurs d'une même variable puisqu'une instanciation de variable par un producteur libère tous les consommateurs suspendus en attente d'une valeur pour cette variable. Nous ne considérons ici que les variables partagées introduites pour garantir la succession car les autres variables partagées éventuelles étant issues de la description des tâches, la synchronisation les concernant est supposée correcte puisque les descriptions de départ le sont.

Aucun risque de blocage permanent ne peut donc se présenter.

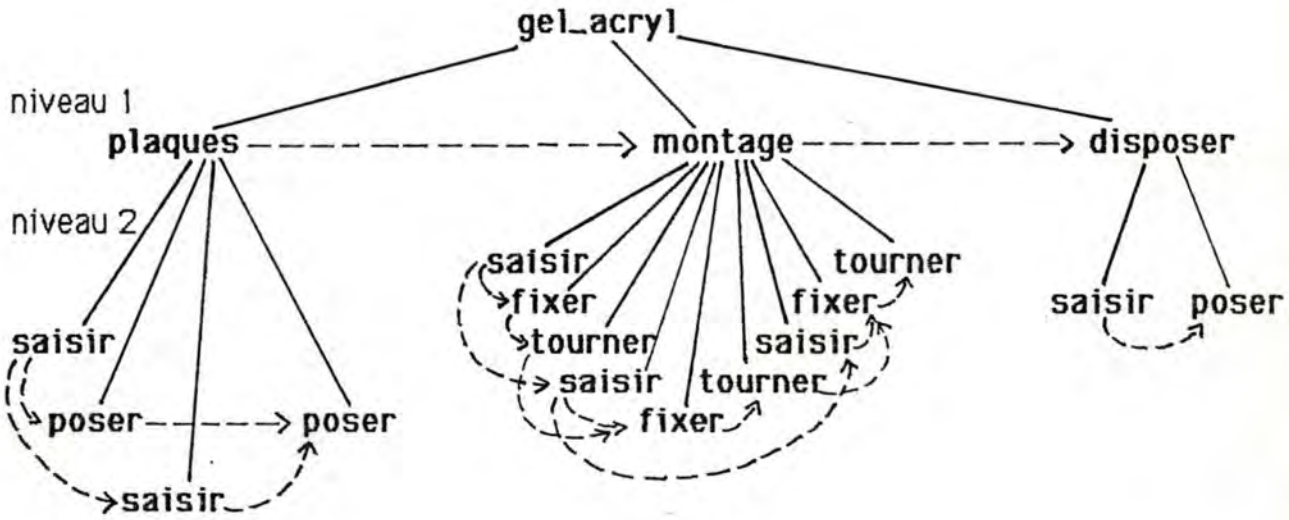
Nous venons de montrer que les trois problèmes pouvant être à l'origine de la non-terminaison du programme GHC ne peuvent se produire. L'exécution du programme ainsi obtenu se terminera donc toujours en un temps fini.

Nous avions vu que les deux propriétés qu'un programme doit vérifier pour être correct sont la validité partielle et la terminaison. Ces deux propriétés étant vérifiées par le programme GHC, celui-ci peut être déclaré correct.

Voyons maintenant une illustration de la méthodologie qui vient d'être présentée sur un exemple complet.

IV.3. Exemple complet : l'application des gels acryliques

L'application concernée est celle dite "des gels acryliques", qui a fait l'objet de l'analyse du chapitre III. Nous donnons pour rappel l'arbre de description des tâches de l'application.



IV.3.1. Descriptions des tâches de l'application

Les descriptions seront données niveau par niveau, en fonction de l'arbre de description présenté ci-dessus.

niveau 1

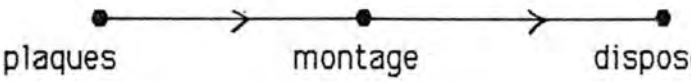
a) enchaînement séquentiel des tâches

L'enchaînement est décrit sous forme d'une clause en Prolog séquentiel.

gel_acryl :- plaques, montage, disposer.

b) contraintes de succession

Les contraintes de succession sont décrites sous forme graphique.



(1)

c) description des sous-tâches

Les sous-tâches n'étant pas primitives, elles sont décrites par les descriptions du niveau 2.

niveau 2

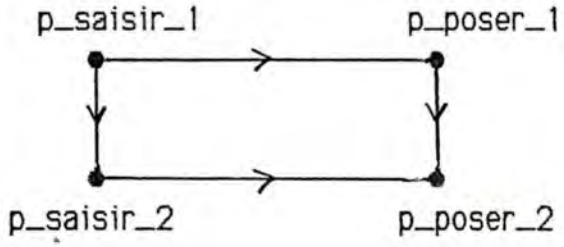
Les trois parties de descriptions (enchaînement, contraintes, description) seront abordées séparément pour chacune des trois sous-tâches à considérer (plaques, montage et disposer).

sous-tâche plaques

a) enchaînement séquentiel des tâches

plaques :- p_saisir_1, p_poser_1, p_saisir_2, p_poser_2.

b) contraintes de succession



(2)

c) description des sous-tâches

Les quatre sous-tâches sont primitives, elles sont donc décrites par identification de la primitive leur correspondant.

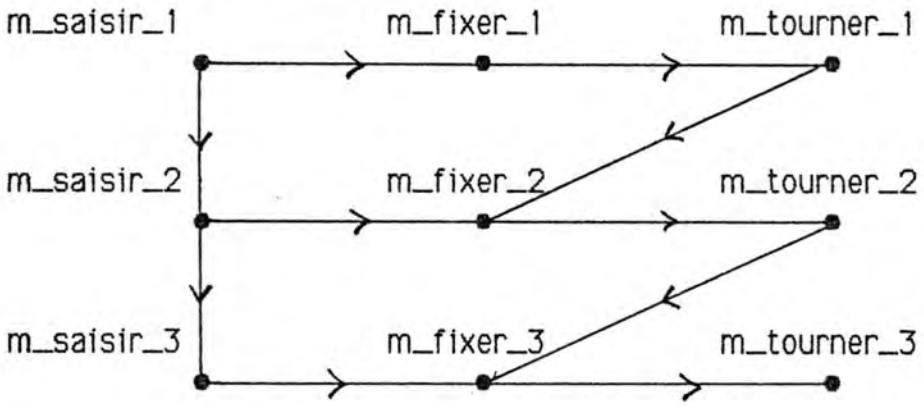
p_saisir_1 :- saisir.
p_saisir_2 :- saisir.
p_poser_1 :- poser.
p_poser_2 :- poser.

sous-tâche montage

a) enchaînement séquentiel des sous-tâches

montage :- m_saisir_1, m_fixer_1, m_tourner_1,
m_saisir_2, m_fixer_2, m_tourner_2,
m_saisir_3, m_fixer_3, m_tourner_3.

b) contraintes de succession



(3)

c) description des sous-tâches

Toutes ces sous-tâches sont primitives.

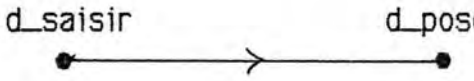
m_saisir_1 :- saisir.
m_saisir_2 :- saisir.
m_saisir_3 :- saisir.
m_fixer_1 :- fixer.
m_fixer_2 :- fixer.
m_fixer_3 :- fixer.
m_tourner_1 :- tourner.
m_tourner_2 :- tourner.
m_tourner_3 :- tourner.

sous-tâche disposer

a) enchaînement séquentiel

disposer :- d_saisir, d_poser.

b) contrainte de succession



(4)

c) description des sous-tâches

Les deux sous-tâches sont primitives.

d_saisir :- saisir.
d_poser :- poser.

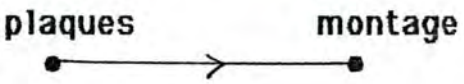
IV.3.2. Vérification des descriptions de tâches

La méthode à suivre pour effectuer cette vérification a été décrite à la section IV.2.3. Elle comporte essentiellement trois étapes :

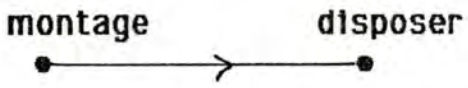
1) vérification des contraintes,
2) vérification de la cohérence entre contraintes et enchaînement séquentiel,
3) vérification des descriptions de tâches.

La *vérification de cohérence des contraintes* peut être ici effectuée par une simple consultation des graphes de contraintes. Nous pouvons constater que les quatres graphes donnés ci-dessus ne comportent pas de circuit. La cohérence des contraintes est donc garantie.

La *vérification de la cohérence entre les contraintes et l'enchaînement séquentiel* doit être effectuée de façon minutieuse. Pour le premier graphe de contraintes (graphe (1)), nous allons considérer les deux arcs séparément.



La sous-tâche 'plaques' étant située à gauche de la sous-tâche 'montage' dans l'enchaînement séquentiel, cette contrainte n'est pas contradictoire avec l'enchaînement.



De la même façon, cette contrainte entre 'montage' et 'disposer' ne contredit pas l'ordre prescrit dans l'enchaînement.

Un raisonnement identique est appliqué pour chaque arc des trois autres graphes (graphes (2), (3) et (4)). On peut constater qu'il n'y a pas d'incohérence entre les contraintes et les enchaînements.

La *vérification de la description des tâches* permet d'affirmer qu'à chaque tâche citée correspond une description.

En conclusion, on peut considérer ces descriptions comme étant correctement rédigées. Elles constituent donc un bon support pour la transformation en un programme GHC.

IV.3.3. Transformation de la description séquentielle des tâches en un programme GHC

Comme expliqué à la section IV.2.4., l'obtention du programme GHC est principalement basée sur la partie des descriptions concernant les contraintes de succession. Chaque contrainte sera gérée par une variable de synchronisation.

Ces contraintes vont être représentées en travaillant niveau par niveau. Pour le premier niveau (graphe (1)), il y a deux contraintes ayant pour origines deux tâches différentes. Il y aura dès lors deux variables partagées dans l'enchaînement des sous-tâches. Une première variable aura le processus correspondant à 'plaques' comme producteur et celui correspondant à 'montage' comme consommateur. Le producteur d'une valeur pour la seconde variable sera le processus correspondant à 'montage' et le consommateur, celui correspondant à 'disposer'. Des arguments seront ajoutés aux prédicats correspondant aux sous-tâches en fonction du schéma de synchronisation producteur/consommateur qui vient d'être décrit. Le prédicat principal de l'application ne devant pas être synchronisé, il n'est pas nécessaire de lui adjoindre un argument supplémentaire. En fonction des principes énoncés au point c de la section IV.2.4., la clause GHC correspondant à la description de la tâche principale 'gel_acryl' sera la suivante :

gel_acryl :- true, !, plaques(P), montage(P,M), disposer(M).

Les descriptions des trois sous-tâches plaques, montage et disposer vont être analysées séparément.

a) description de 'plaques'

Le processus correspondant à 'plaques' est le producteur de valeur pour la variable P. Cette variable restera non instanciée jusqu'au niveau des primitives (cfr point b de la section IV.2.4.).

La description de 'plaques' est basée sur une partie du deuxième niveau de l'arbre de l'application. Cette partie correspond au graphe de contraintes (2) (section IV.3.1.). La représentation de ce graphe nécessitera l'utilisation de trois variables de synchronisation (une variable par tâche différente qui est à l'origine d'une contrainte). Ces trois variables, ainsi que leurs producteurs et consommateurs respectifs, sont représentés dans le tableau ci-dessous.

variables	producteurs	consommateurs
V1	p_saisir_1	p_poser_1 p_saisir_2
V2	p_poser_1	p_poser_2
V3	p_saisir_2	p_poser_2

On peut remarquer que seule la primitive 'p_poser_2' n'est origine d'aucune contrainte. C'est dès lors à son niveau que sera instanciée la variable chargée de signaler la fin de l'exécution de la tâche 'plaques'. Cette variable lui sera donc ajoutée comme argument, en plus des variables nécessaires pour la synchronisation avec les autres sous-tâches de son niveau. La transformation de l'enchaînement tel qu'il avait été défini en Prolog séquentiel dans les descriptions de tâches sera effectuée sur base de la démarche présentée aux points b et c de la section IV.2.4. Le processus 'plaques' étant producteur de la variable de synchronisation correspondant à son argument, cette variable est non instanciée dans la tête de la clause de 'plaques'. La garde de cette clause GHC est vide, et sera donc représentée par le prédicat 'true'. Le corps de la clause reprendra les prédicats des quatre sous-tâches, auxquels on aura ajouté les arguments correspondant aux variables de synchronisation dont l'utilisation a été décrite dans le tableau ci-dessus. Le code GHC de la clause de la procédure 'plaques' sera dès lors le suivant :

```
plaques(P) :- true, !, p_saisir_1(V1), p_poser_1(V1,V2),
                       p_saisir_2(V1,V3), p_poser_2(V2,V3,P).
```

Les sous-tâches de la tâche plaques étant primitives, les corps des clauses leur correspondant seront formés d'appels aux primitives qui doivent être exécutées par les appareils. C'est au niveau des primitives que les instanciations des variables de synchronisation sont effectuées (cfr section III.4.). Il n'est pas nécessaire dans le cadre de cette transformation de décrire complètement ces primitives. En effet, elles appartiennent à la partie fixe du programme GHC, et sont donc les mêmes pour toute application. Nous allons dès lors nous limiter à la présentation des valeurs qu'elles assignent aux variables de synchronisation. Ces primitives, tout comme nous l'avons vu à la section III.1., sont au nombre de quatre. Leur nom et la valeur qu'elles assignent à la variable de synchronisation sont repris dans le tableau ci-dessous.

primitives	**valeurs assignées à la variable de synchronisation**
saisir	fin_saisir
poser	fin_poser
fixer	fin_fixer
tourner	fin_tourner

La synchronisation entre les quatre sous-tâches de la tâche 'plaques' est effectuée selon le principe suivant. Une sous-tâche consommateur d'une variable à laquelle une valeur est assignée par la sous-tâche p_saisir_1 (par exemple) devra citer dans l'argument de tête de clause correspondant à cette variable la valeur 'fin_saisir'. En effet, p_saisir_1 est producteur et fait appel à la primitive 'saisir', c'est donc la valeur 'fin_saisir' qui sera assignée à la variable de synchronisation. En citant cette valeur dans leur tête de clause, les consommateurs seront suspendus tant que la sous-tâche producteur (ici, p_saisir_1) ne sera pas terminée (n'aura pas instancié la variable partagée). Le même principe est appliqué pour toutes les variables partagées représentant les relations producteur/consommateur.

Sur base de la description de la tâche 'plaques' et du tableau de synchronisation décrivant les relations entre variables, producteurs et consommateurs, une description en GHC des quatre sous-tâches primitives peut être proposée :

```
p_saisir_1(X) :- true, !, saisir(X).
p_poser_1(fin_saisir,X) :- true, !, poser(X).
p_saisir_2(fin_saisir,X) :- true, !, saisir(X).
p_poser_2(fin_saisir,fin_poser,P) :- true, !, poser(P).
```

b) description de "montage"

Le processus correspondant à 'montage' est un consommateur de la variable P, et un producteur pour la variable M (voir description de la tâche principale 'gel_acryl' au début de la section IV.3.3.). La valeur que la variable de synchronisation P reçoit à la fin de l'exécution de la tâche plaques est "fin_poser" (car P est instanciée par la primitive 'poser'). Cette valeur sera citée explicitement dans la tête de la clause correspondant à la description de la tâche 'montage', de façon à effectuer la synchronisation entre processus, comme expliqué à la section IV.2.4., point b. Le graphe de contraintes (3) (cfr section IV.3.1.) nous donne les caractéristiques des sous-tâches décrivant cette tâche. L'implémentation de ce graphe se fera en utilisant 8 variables de synchronisation (une variable par contrainte d'origine différente). Les relations entre ces variables et les sous-tâches producteurs et consommateurs sont présentées dans le tableau ci-dessous.

variables	**producteurs**	**consommateurs**
V1	m_saisir_1	m_fixer_1 m_saisir_2
V2	m_fixer_1	m_tourner_1
V3	m_tourner_1	m_fixer_2
V4	m_saisir_2	m_fixer_2 m_saisir_3
V5	m_fixer_2	m_tourner_2
V6	m_tourner_2	m_fixer_3
V7	m_saisir_3	m_fixer_3
V8	m_fixer_3	m_tourner_3

Seule la sous-tâche 'm_tourner_3' n'est origine d'aucune contrainte. Ce sera donc elle qui effectuera l'instanciation de la variable de fin d'exécution de 'montage'. La démarche suivie pour obtenir la clause de la tâche 'montage' est tout-à-fait similaire à celle utilisée pour la tâche 'plaques'. Elle ne sera dès lors pas décrite ici. Le code GHC obtenu pour la description de la tâche 'montage' est le suivant :

```
montage(fin_poser,M) :- true, !,
        m_saisir_1(V1), m_fixer_1(V1,V2),
        m_tourner_1(V2,V3), m_saisir_2(V1,V4),
        m_fixer_2(V3,V4,V5), m_tourner_2(V5,V6),
        m_saisir_3(V4,V7), m_fixer_3(V6,V7,V8),
        m_tourner_3(V8,M).
```

Les sous-tâches de la tâche 'montage' étant primitives, chacune d'elles est définie selon le même principe que celui adopté pour la description des sous-tâches primitives de la tâche 'plaques'. Le code GHC correspondant à ces tâches primitives est donné ci-dessous.

```
m_saisir_1(X) :- true, !, saisir(X).
m_fixer_1(fin_saisir,X) :- true, !, fixer(X).
m_tourner_1(fin_fixer,X) :- true, !, tourner(X).
m_saisir_2(fin_saisir,X) :- true, !, saisir(X).
m_fixer_2(fin_saisir,fin_tourner,X) :- true, !, fixer(X).
m_tourner_2(fin_fixer,X) :- true, !, tourner(X).
m_saisir_3(fin_saisir,X) :- true, !, saisir(X).
m_fixer_3(fin_saisir,fin_tourner,X) :- true, !, fixer(X).
m_tourner_3(fin_fixer,M) :- true, !, tourner(M).
```

c) description de "disposer"

Le processus correspondant à la tâche 'disposer' est consommateur de la variable partagée M, et n'est producteur de valeur pour aucune variable car il n'est origine d'aucune contrainte de succession.

Comme c'est la sous-tâche 'm_tourner_3' qui instancie la variable de synchronisation M, la valeur attribuée à cette variable à la fin de l'exécution de la tâche 'montage' sera "fin_tourner". C'est cette valeur qui sera spécifiée dans la tête de la clause correspondant à la tâche 'disposer'.

Le graphe de contraintes (4) décrit l'enchaînement des deux sous-tâches de la tâche 'disposer'. Une seule variable de synchronisation sera nécessaire : le producteur en sera la sous-tâche 'd_saisir' et le consommateur, la sous-tâche 'd_poser'.

Nous avons vu au point a ci-dessus (description de la tâche 'plaques') que toute primitive instanciait en fin d'exécution la variable de synchronisation avec laquelle elle est invoquée. La sous-tâche 'd_poser' ne doit produire de valeur pour aucune variable car :

1) elle est seulement consommateur pour la synchronisation nécessaire à son niveau,

2) la tâche de niveau supérieur dont elle dépend (disposer) est seulement consommateur pour la synchronisation à son niveau,

3) la tâche dont 'disposer' dépend est la tâche principale 'gel_acryl' (ne nécessitant pas de synchronisation).

Cette sous-tâche 'd_poser' doit tout de même faire appel à la primitive 'poser', de façon à ce que la tâche soit exécutée par l'appareil. Cette primitive 'poser' va donc être appellée avec un argument correspondant à une variable de synchronisation qui ne sera pas utilisée par d_poser : cet argument sera instancié inutilement. On aurait pu, tout comme dans le code de l'application des gels acryliques donné en annexe C, spécifier au niveau de la tâche 'disposer' une variable de fin d'exécution qui servirait à une synchronisation avec une tâche ultérieure non encore décrite. La tâche 'disposer 'serait alors caractérisée comme consommateur de la variable qu'elle partage avec la tâche 'montage' (M), et comme producteur d'une variable partagée avec une autre tâche qui pourrait être décrite par la suite (soit D).

Comme la sous-tâche 'd_poser' n'est pas origine de l'unique contrainte apparaissant dans la description de la tâche 'disposer', c'est elle qui devrait instancier la variable de synchronisation D.

Cette seconde solution semble être plus claire, car à partir du moment où la description de l'application doit être modifiée par l'ajout d'une nouvelle tâche, il ne faut rien changer à la description de la tâche disposer. Avec la première solution, il faudrait dans ce cas aller modifier le code GHC de la tâche 'disposer'. La seconde solution offre donc une facilité de maintenance, l'application étant plus facile à modifier et à étendre. C'est elle qui sera retenue pour la description de la tâche 'disposer'.

La méthode déjà appliquée pour obtenir les procédures décrivant les tâches 'plaques' et 'montage' sera à nouveau utilisée ici. L'implémentation de la seconde solution proposée ci-dessus nécessite l'ajout d'un argument pour la procédure 'disposer', par rapport à la description de la tâche principale 'gel_acryl' effectuée au début de la section IV.3.3. Cet argument correspond à la variable de synchronisation supplémentaire D.

Le code obtenu pour la tâche 'disposer' et les deux sous-tâches primitives 'd_saisir' et 'd_poser' est dès lors le suivant :

```
disposer(fin_tourner,D) :- true, !, d_saisir(V), d_poser(V,D).
d_saisir(X) :- true, !, saisir(X).
d_poser(fin_saisir,D) :- true, !, poser(D).
```

IV. . Proposition d'extensions

Le code GHC obtenu par la transformation de la description séquentielle des tâches ne correspond pas exactement au code proposé au chapitre III pour l'application robotique. En effet, la transformation telle qu'elle a été proposée à la section IV.2.4. ne comporte pas de boucles.

Une extension possible serait de pouvoir générer un code GHC construit avec des boucles, tout comme elles sont décrites à la section III.5.2. En effet, le code ainsi obtenu serait beaucoup plus lisible qu'un code dans lequel toutes les tâches primitives sont invoquées une à une.

Il faudrait pour ce faire disposer de descriptions de tâches dans lesquelles les boucles seraient déjà présentes. Remarquons que cela ne va pas à l'encontre des descriptions proposées à la section IV.2.2. car ces boucles ne sont pas l'exclusivité d'une exécution parallèle. Il pourrait dès lors être logique de décrire une application en Prolog séquentiel avec des boucles.

Une méthodologie différente devrait alors être employée pour transformer les fragments de code Prolog pouvant contenir des définitions de boucles en un programme GHC.

Une telle approche n'a pas été retenue dans la méthodologie proposée ici, de façon à favoriser une description plus élémentaire visant à mettre l'accent sur le mécanisme de synchronisation des différentes tâches de l'application.

La méthodologie de transformation proposée ici n'aborde en rien la partie des applications robotiques concernant la gestion des appareils. La raison en est que cette partie du programme GHC est statique pour un certain environnement de travail. Elle ne doit pas être modifiée pour les différentes applications robotiques exécutées dans l'environnement concerné.

Il pourrait cependant être intéressant de prendre en considération cette partie du programme GHC, de façon à en vérifier la validité. On pourrait dès lors essayer de trouver une méthode permettant de vérifier si cette gestion des appareils est correctement codée en GHC. Sans cette méthode, une erreur dans la description de l'utilisation des appareils ne pourrait être détectée qu'à l'exécution du programme de l'application, par la présence d'un échec cyclique (voir point c de la section

II.5.3.). Or, nous avons vu au début de ce chapitre que les tests de programmes ne pouvaient en rien prouver l'absence d'erreurs dans un programme.

Le développement d'une méthodologie vérifiant la validité de la partie 'statique' des applications robotiques pourrait dès lors être une orientation intéressante à suivre pour des recherches ultérieures.

Conclusion

Cette étude du langage GHC peut être divisée en deux parties : la première effectue la comparaison entre GHC et les deux langages logiques parallèles Parlog et Concurrent Prolog; la seconde évalue l'utilisation de GHC dans le contexte particulier des applications robotiques.

La *première partie* de ce travail est assez théorique car elle repose sur les définitions des concepts liés au parallélisme et les descriptions synthétiques des trois langages logiques parallèles considérés.

Les résultats obtenus par la comparaison de Parlog, Concurrent Prolog et GHC n'ont pas permis de différencier nettement ces trois langages. Peut-être serait-il dès lors intéressant de définir et d'analyser de façon précise leurs sémantiques procédurales respectives, toutes trois relativement complexes. Cependant, l'ampleur d'un tel projet nous a incités à ne considérer que les principaux avantages et inconvénients propres à chacun des trois langages.

Le développement d'un interpréteur de GHC semblait être l'occasion idéale pour comparer minutieusement GHC avec le langage Concurrent Prolog. Grâce à la ressemblance existant entre ces deux langages, l'interpréteur écrit par Shapiro pour Concurrent Prolog (cfr [Sh83]) a largement servi de base à la conception de celui proposé ici pour GHC. Cependant, les restrictions apportées aux implémentations de GHC et de Concurrent Prolog au niveau de la simulation du parallélisme n'ont pas permis de comparer ces deux langages dans leur ensemble. Des caractéristiques telles que le parallélisme tête-garde-corps (propre à GHC) et la gestion d'environnements multiples (propre à Concurrent Prolog) n'ayant pas été implémentées, certaines des différences existant entre les deux langages sont masquées par les interpréteurs développés. Des interpréteurs ne limitant en rien le parallélisme permettraient d'étudier de façon plus approfondie les relations existant entre ces deux langages. Cette orientation pour des recherches futures semblant être liée au développement d'un matériel adéquat, elle présentait un aspect qui sortait du contexte de cette étude.

La *seconde partie* de ce travail présente un aspect plus pratique et plus concret de l'étude du langage GHC.

La description d'une application robotique complète nous a montré comment GHC pouvait être employé pour concevoir des logiciels de contrôle robotique. Cette utilisation de GHC présente un inconvénient au niveau de la clarté du principe de synchronisation employé. Le manque de simplicité de ce principe rendait dès lors particulièrement intéressante une proposition d'approche systématique de la conception de la partie des applications concernant la synchronisation.

Un autre aspect de cette méthodologie de conception d'applications robotiques est la possibilité qui est offerte de transformer systématiquement des

applications séquentielles en leur ajoutant le parallélisme de GHC. La validité du programme GHC ainsi obtenu est garantie.

Cette étude de GHC dans le domaine de la robotique ne constitue pas, de par la (quasi)-absence de travaux apparentés réalisés pour Parlog et Concurrent Prolog, une base de comparaison entre GHC et ces deux autres langages logiques parallèles. Elle représente néanmoins une étude originale de GHC. Il serait dès lors intéressant, dans le contexte de la poursuite des recherches sur le langage GHC, d'effectuer une telle approche dans le cadre d'autres domaines d'application. Les voies ouvertes pour des travaux ultérieurs restent nombreuses.

Bibliographie

[Beg70] Claude BERGE
Graphes et hypergraphes
Dunod, Paris, 1970.

[Bet86] Philippe BERTHET
Un environnement d'aide à la programmation de robot : la couche objet.
mémoire, Institut d'Informatique,
Facultés Universitaires Notre-Dame de la Paix, Namur, 1986.

[Br85] Michaël BRADY
Artificial Intelligence and Robotics.
pp 79-121,
Artificial Intelligence, number 26, 1985.

[BrGr84] Krysia BRODA & Steve GREGORY
Parlog for discrete event simulation.
Proceedings of the second international logic programming conference,
Uppsala University, July 1984.

[BrDe82] Randal E. BRYANT & Jack B. DENNIS
Concurrent Programming.
M. Madegawa & L.A.Belady Editors, Operating Systems Engineering,
pp 426-452,
Springer-Verlag, 1982.

[ClGr86] Keith CLARK & Steve GREGORY
Parlog : parallel Programming in Logic.
ACM Transactions on Programming Languages and Systems,
Volume 8, Number 1, January 1986.

[ClTa77] Keith L. CLARK & Sten-Åke TÄRNLUND
A first order theory of data and programs.
pp 939 - 944,
Information Processing 77, B. Gilcrist, Editor,
IFIP, North-Holland, 1977.

[ClMe81] W.F. CLOCKSIN & C.S. MELLISH
Programming in Prolog.
Springer-Verlag, 1981.

[Co81] CORNAFION
Systèmes informatiques répartis.
Dunod, 1981.

[CrMi84] J.A. CRAMMOND & C.D.F. MILLER
An architecture for parallel logic languages.
Proceedings of the second international logic programming conference,
Uppsala University, July 1984.

[Dv84] Luc DE VOS
Studie en verwezenlijking van de taal Concurrent Prolog.
Eindwerk, Faculteit der Toegepaste Wetenschappen,
Departement Computerwetenschappen,
Katholieke Universiteit Leuven, 1984.

[Di71] Edsger W. DIJKSTRA
Hierarchical Ordering of Sequential Processes.
pp 115-138,
Acta Informatica, Volume 1, 1971.

[Di75] Edsger W. DIJKSTRA
Guarded Commands, Nondeterminacy and Formal Derivation of Programs.
pp 453-457,
Communications of the ACM, Volume 18, Number 8, August 1975.

[Fi] Jean FICHEFET
Théorie des graphes.
notes de cours, Institut d'Informatique,
Facultés Universitaires Notre-Dame de la Paix, Namur.

[Ha73] Per BRINCH HANSEN
Concurrent Programming Concepts.
pp 223-245,
Computing Surveys, Volume 5, Number 4, December 1973.

[Ho78] C.A.R. HOARE
Communicating Sequential Processes.
pp 666-677,
Communications of the ACM, Volume21, Number 8, August 1978.

[Hog84] Chistopher John HOGGER
Introduction to logic programming.
Academic Press, 1984.

[Ko74] Robert KOWALSKI
Predicate logic as programming language.
Information Processing 74,
North-Holland Publishing Company, 1974.

[Ko79a] Robert KOWALSKI
Logic for Problem Solving.
North-Holland, 1979.

[Ko79b] Robert KOWALSKI
Algorithm = Logic + Control.
pp 424 - 435,
Communications of the ACM, Volume 22, Number 7, July 1979.

[Le84] Jacob LEVY
A unification algorithm for Concurrent Prolog.
Proceedings of the second international logic programming conference,
Uppsala University, July 1984.

[Ll84] J. W. LLOYD
Foundations of logic programming.
Symbolic computation, Springer-Verlag, 1984.

[Mi84] C. MIEROWSKY
Design and implementation of Flat Concurrent Prolog.
Msc Thesis, Dept. of Applied Mathematics,
Weizmann Institute of Science, Rehovot, November 1984.

[Ni71] Nils J. NILSSON
Problem solving methods in Artificial Intelligence.
Mc Graw-Hill, 1971.

[Ro69] B. ROY
Algèbre moderne et théorie des graphes
Dunod, Paris, 1969.

[Sh83] Ehud SHAPIRO
A subset of Concurrent Prolog and its interpreter.
Technical Report TR-003,
ICOT, 1983.

[TaFu83] Akikazu TAKEUCHI & Kouichi FURUKAWA
Interprocess Communication in Concurrent Prolog.
LP workshop, 1983.

[Ue85a] Kazunori UEDA
Guarded Horn Clauses.
ICOT Technical Report TR-103,
C&C Systems Research Laboratories, NEC Corporation, June 1985.

[Ue85b] Kazunori UEDA
Concurrent Prolog Re-Examined.
Draft of the ICOT technical report TR-102,
C&C Systems Research Laboratories, NEC Corporation, June 1985.

[UeCh85] Kazunori UEDA & Takashi CHIKAYAMA
Concurrent Prolog compiler on top of Prolog.
pp 119 - 126,
IEEE, 1985.

[Va85] Claude VANHEMELRYCK
Construction d'un environnement d'aide à la programmation d'un micro-robot.
Mémoire, Institut d'Informatique,
Facultés Universitaires Notre-Dame de la Paix, Namur, 1985.

[Vla86] Axel van LAMSWEERDE
A Transformation Technique for making programs more parallel.
Report RP 1/86, Institut d'Informatique,
Facultés Universitaires Notre-Dame de la Paix, Namur, February 1986.

Annexe A

Code d'un interpréteur GHC avec trace

1) Code de l'interpréteur

```
solve(Goals) :- schedule(Goals,X,X,Head,[cycle|Tail]),
        ghc_trace('SCHEDULED',Goals,Head),
        ghc_solve(Head,Tail,échec),
        ghc_trace('SOLVED',Goals).

schedule([],Head,Tail,Head,Tail).
schedule([Goal|List],Head,[Goal|Tail],Newhead,Newtail) :-
            schedule(List,Head,Tail,Newhead,Newtail).

ghc_solve([cycle],[],_) :- !.
ghc_solve([cycle|Head],[cycle|Tail],pas_échec) :- !,
                        ghc_solve(Head,Tail,échec).
ghc_solve([cycle|Head],Tail,échec) :-
         ghc_trace('ECHEC','LOCKED_PROCESSES',Head), !, fail.
ghc_solve([G|Head],Tail,E) :- builtin(G),
                  call(G), !,
                  ghc_trace('SYSTEM_SOLVED',G,Head),
                  ghc_solve(Head,Tail,pas_échec).
ghc_solve([G|Head],[G|Tail],E) :- builtin(G), !,
                    ghc_trace('SUSPENDED',G,Head),
                    ghc_solve(Head,Tail,E).
ghc_solve([G|Head],Tail,E) :- reduce(G,R), !,
                  schedule(R,Head,Tail,Newhead,Newtail),
                  ghc_trace('SCHEDULED',R,Newhead),
                  ghc_solve(Newhead,Newtail,pas_échec).
ghc_solve([G|Head],[G|Tail],E) :- ghc_trace('SUSPENDED',G,Head),
                                  ghc_solve(Head,Tail,E).

reduce(Goal,Reduced) :- guarded_clause(Goal,Guard,Reduced), solve(Guard),
                        ghc_trace('REDUCED',Goal,Reduced).
```

```
guarded_clause(Goal,Guard,Body) :- find_clause(Goal,Clause),
                                   find_guard(Clause,Guard,Body).

find_clause(Goal,Clauselist) :-
        functor(Goal,Functor,Nbarg),
        functor(Newgoal,Functor,Nbarg),
        clause(Newgoal,Clause),
        unify(Goal,Newgoal),
        liste(Clause,Clauselist).

find_guard(['!'|Body],[],Body) :- !.
find_guard([Goal|Tail],[Goal|Guard],Body) :- find_guard(Tail,Guard,Body).

unify(X,Y) :- var(Y), !, X = Y.
unify(X,Y) :- var(X), !, fail.
unify([Elt1|List1],[Elt2|List2]) :- !, unify(Elt1,Elt2), unify(List1,List2).
unify([],[]) :- !.
unify(X,Y) :-  X =.. [F|ArgX], Y =.. [F|ArgY], unify(ArgX,ArgY).

liste(Terms,List) :- liste(Terms,X,X,List,[]).
liste(Conj,Head,Tail,Head2,Tail2) :-
                Conj =.. [',',H|[T]]], !,
                liste(H,Head,Tail,Head1,Tail1),
                liste(T,Head1,Tail1,Head2,Tail2).
liste(Conj,Head,[Conj|Tail],Head,Tail).

ghc_trace(X,Goals) :-  'P'(X), tab(1), 'P'(':'), tab(1), 'PP'(Goals).
ghc_trace(X,Goal,Head) :-  'P'(X), tab(1), 'P'(':'), tab(1), 'PP'(Goal),
                           ecrire_liste(Head).

ecrire_liste(X) :- var(X).
ecrire_liste([]).
ecrire_liste([Goal|List]) :- tab(10), 'PP'(Goal), ecrire_liste(List).
```

2) prédicats prédéfinis

```
builtin(true).
builtin(X = Y).
```

3) Code GHC du programme de fusion

```
merge([Xs|X],Y,Z) :- true, !, Z = [Xs|W],
                     merge(X,Y,W).
merge(X,[Ys|Y],Z) :- true, !, Z = [Ys|W],
                     merge(X,Y,W).
merge([],Y,Z) :- true, !, Z = Y.
merge(X,[],Z) :- true, !, Z = X.
```

4) Trace de la résolution de solve([merge([a,b],[c],L)])

```
SCHEDULED : ((merge@<(a b) (c) _360>))
        (merge@<(a b) (c) _360>)
        cycle
SCHEDULED : (true)
        true
        cycle
SYSTEM_SOLVED : true
        cycle
SOLVED : (true)
REDUCED : (merge@<(a b) (c) _500>)
        (=@<_500 (a|_563)>)
        (merge@<(b) (c) _563>)
SCHEDULED : ((=@<_500 (a|_563)>) (merge@<(b) (c) _563>))
        cycle
        (=@<_500 (a|_563)>)
        (merge@<(b) (c) _563>)
SYSTEM_SOLVED : (=@<(a|_563) (a|_563)>)
        (merge@<(b) (c) _563>)
        cycle
SCHEDULED : (true)
        true
        cycle
SYSTEM_SOLVED : true
        cycle
SOLVED : (true)
REDUCED : (merge@<(b) (c) _1002>)
```

```
        (=@<_1002 (b|_1065)>)
        (merge@<() (c) _1065>)
SCHEDULED : ((=@<_1002 (b|_1065)>) (merge@<() (c) _1065>))
        cycle
        (=@<_1002 (b|_1065)>)
        (merge@<() (c) _1065>)
SYSTEM_SOLVED : (=@<(b|_1065) (b|_1065)>)
        (merge@<() (c) _1065>)
        cycle
SCHEDULED : (true)
        true
        cycle
SYSTEM_SOLVED : true
        cycle
SOLVED : (true)
REDUCED : (merge@<() (c) _1504>)
        (=@<_1504 (c|_1568)>)
        (merge@<() () _1568>)
SCHEDULED : ((=@<_1504 (c|_1568)>) (merge@<() () _1568>))
        cycle
        (=@<_1504 (c|_1568)>)
        (merge@<() () _1568>)
SYSTEM_SOLVED : (=@<(c|_1568) (c|_1568)>)
        (merge@<() () _1568>)
        cycle
SCHEDULED : (true)
        true
        cycle
SYSTEM_SOLVED : true
        cycle
SOLVED : (true)
REDUCED : (merge@<() () _2007>)
        (=@<_2007 ()>)
SCHEDULED : ((=@<_2007 ()>))
        cycle
        (=@<_2007 ()>)
SYSTEM_SOLVED : (=@<() ()>)
        cycle
SOLVED : ((merge@<(a b) (c) (a b c)>))
L =  [a,b,c]
```

Annexe B

Code d'un interpréteur GHC avec non-déterminisme-ET

```
solve(Goals) :- schedule(Goals,[],List),
                ghc_solve(List).

schedule([],List,List).
schedule([A|B],List,Newlist) :- length(List,L), rando(L,Loc),
                                insert(A,List,List1,Loc),
                                schedule(B,List1,Newlist).

ghc_solve([]) :- !.
ghc_solve([Goal|List]) :- builtin(Goal),
                          call(Goal), !,
                          ghc_solve(List).
ghc_solve([Goal|List]) :- builtin(Goal),
                          schedule([Goal],List,Newlist),
                          ghc_solve(Newlist).
ghc_solve([Goal|List]) :- reduce(Goal,Reduced), !,
                          schedule(Reduced,List,Newlist),
                          ghc_solve(Newlist).
ghc_solve([Goal|List]) :- schedule([Goal],List,Newlist),
                          ghc_solve(Newlist).

reduce(Goal,Reduced) :- guarded_clause(Goal,Guard,Reduced), solve(Guard).

guarded_clause(Goal,Guard,Body) :- find_clause(Goal,Clause),
                                   find_guard(Clause,Guard,Body).

find_clause(Goal,Clauselist) :-
            functor(Goal,Functor,Nbarg),
            functor(Newgoal,Functor,Nbarg),
            clause(Newgoal,Clause),
            unify(Goal,Newgoal),
            liste(Clause,Clauselist).

find_guard(['!'|Body],[],Body) :- !.
find_guard([Goal|Tail],[Goal|Guard],Body) :- find_guard(Tail,Guard,Body).
```

```
unify(X,Y) :- var(Y), !, X = Y.
unify(X,Y) :- var(X), !, fail.
unify([Elt1|List1],[Elt2|List2]) :- !, unify(Elt1,Elt2), unify(List1,List2).
unify([],[]) :- !.
unify(X,Y) :-  X =.. [F|ArgX], Y =.. [F|ArgY], unify(ArgX,ArgY).

liste(Terms,List) :- liste(Terms,X,X,List,[]).
liste(Conj,Head,Tail,Head2,Tail2) :-
                        Conj =.. [','|[H|[T]]], !,
                        liste(H,Head,Tail,Head1,Tail1),
                        liste(T,Head1,Tail1,Head2,Tail2).
liste(Conj,Head,[Conj|Tail],Head,Tail).

length(List,L) :- length(List,L,0).
length([],L,L).
length([A|B],L,Lt) :- L1 is Lt + 1, length(B,L,L1).

rando(0,0).
rando(L,Loc) :- retract(seed(S)),
                A is S mod L, 'ABS'(A,B), Loc is B + 1,
                New is (125 * S + 1) mod 4096, 'ABS'(New,Newseed),
                asserta(seed(Newseed)), !.

insert(A,L,[A|L],0) :- !.
insert(A,[H|L],[H|L2],Loc) :- Nloc is Loc - 1, insert(A,L,L2,Nloc).
```

Annexe C

Code de l'application des gels acryliques

niveau 1

```
gel_acryl :- init_appareils, !, plaques(P), montage(P,M), disposer(M,D).
```

niveau 2

```
plaques(Y) :- true, !, empiler(2,plaque,distrib_plaques, site_montage,Y).

montage(fin_empiler,Y) :- true, !,
    attacher(3,pince, distrib_pinces, site_montage,assemblage,90,Y).

disposer(fin_attacher,Y) :- true, !,
    empiler(1,assemblage,site_montage,site_stockage,Y).
```

niveau 3

```
empiler(N,Objet,Site1,Site2,Y) :- true, !,
    e_empiler(N,Objet,Site1,Site2,fin_saisir,fin_poser,Y).

attacher(N,Objet,Site1,Site2,Résultat,Degré,Y) :- true, !,
    a_attacher(N,Objet,Site1,Site2,Résultat,Degré,fin_saisir,fin_tourner,Y).


e_empiler(N,Objet,Site1,Site2,S_in,P_in,Y) :- N > 0, M is N - 1, !,
    saisir(Objet,Site1,Rob,S_in,S_out),
    poser(Objet,Site2,Rob,P_in,S_out,P_out),
    e_empiler(M,Objet,Site1,Site2,S_out,P_out,Y).
e_empiler(0,Objet,S1,S2,fin_saisir,fin_poser,Y) :- true, !, Y = fin_empiler.

a_attacher(N,Objet,Site1,Site2,Résultat,Degré,S_in,C_in,Y) :- N > 0, M is N - 1, !,
    placer(Objet,Site1,Site2,S_in,C_in,S_out,F_out),
    tourner(Résultat,Degré,F_out,C_out),
    a_attacher(M,Objet,Site1,Site2,Résultat,Degré,S_out,C_out,Y).
a_attacher(0,Objet,Site1,Site2,Résultat,Degré,fin_saisir,fin_tourner,Y) :-
    true, !, Y = fin_attacher.
```

```
placer(Objet,Site1,Site2,S_in,C_in,Y1,Y2) :- true, !,
     saisir(Objet,Site1,Rob,S_in,Y1),
     fixer(Objet,Site2,Rob,C_in,Y1,Y2).


saisir(Objet,Site,Rob,fin_saisir,Y) :- true, !, robot([saisir,Objet,Site],Rob,Y).

fixer(Objet,Site,Rob,fin_tourner,fin_saisir,Y) :- true, !,
                                    robot([fixer,Objet,Site],Rob,Y).

poser(Objet,Site,Rob,fin_poser,fin_saisir,Y) :- true, !,
                                   robot([poser,Objet,Site],Rob,Y).

tourner(Objet,Degré,fin_fixer,Y) :- prio_concept(X), !,
                       appareil_concept(X,[tourner,Objet,Degré],Y).
```

gestion des appareils

```
appareil_concept(robot,Tâche,Y) :- true, !, robot(Tâche,Rob,Y).
appareil_concept(tourneur,Tâche,Y) :- true, ! , tourneur(Tâche,_,Y).

robot(Tâche,rob1,Y) :- true, !, rob1(Tâche,_,Y).
robot(Tâche,rob2,Y) :- true, !, rob2(Tâche,_,Y).
robot(Tâche,Rob,Y) :- var(Rob), prio_phys(rob1,rob2,X), !,
                        appareil_phys(X,Tâche,Rob,Y).
tourneur(Tâche,_,Y) :- true, !, tourn1(Tâche,_,Y).

rob1(T,Rob,Y) :- true, !, exec(rob1,T,Y), Rob = rob1.
rob2(T,Rob,Y) :- true, ! , exec(rob2,T,Y), Rob = rob2.
tourn1(T,_,Y) :- true, !, exec(tourn1,T,Y).

appareil_phys(rob1,T,Rob,Y) :- true, !, rob1(T,Rob,Y).
appareil_phys(rob2,T,Rob,Y) :- true, !, rob2(T,Rob,Y).

prio_concept(X) :- disponible(tourn1), !, X = tourneur.
prio_concept(X) :- not(disponible(tourn1)), !, X = robot.
```

```
prio_phys(A,B,X) :- disponible(A), not(disponible(B)), !, X = A.
prio_phys(A,B,X) :- disponible(B), not(disponible(A)), !, X = B.
prio_phys(A,B,X) :- disponible(A), disponible(B),
                    'MYESNO'([A,plus,prioritaire,que,B]), !, X = A.
prio_phys(A,B,X) :- disponible(A), disponible(B),
                    'MYESNO'([B,plus,prioritaire,que,A]), !, X = B.

init_appareils :- true, !, assert(état(rob1,en_service)), assert(état(rob1,libre)),
                           assert(état(rob2,en_service)), assert(état(rob2,libre)),
                           assert(état(tourn1,en_service)), assert(état(tourn1,libre)).
```

interface couche objet

```
exec(X,[saisir,Objet,Site],Y) :- occuper(X),
                          'PP'(saisir,Objet,en,Site,avec,X),
                          Y = fin_saisir.
exec(X,[fixer,Objet,Site],Y) :-  état(X,en_service),
                                 'PP'(fixer,Objet,en,Site,avec,X),
                                 libérer(X), Y = fin_fixer.
exec(X,[poser,Objet,Site],Y) :-  état(X,en_service),
                                 'PP'(poser,Objet,en,Site,avec,X),
                                 libérer(X), Y = fin_poser.
exec(X,[tourner,Objet,Degré],Y) :- occuper(X),
                                'PP'(tourner,Objet,Degré,degrés,avec,X),
                                libérer(X), Y = fin_tourner.

occuper(X) :- état(X,en_service), retract(état(X,libre)), assert(état(X,occupé)).
libérer(X) :- état(X,en_service), retract(état(X,occupé)), assert(état(X,libre)).
```

prédicats prédéfinis

```
builtin(true).
builtin(X = Y).
builtin(var(X)).
builtin(assert(X)).
builtin(exec(X,Y,Z)).
builtin(X > Y).
builtin(X is Y).
builtin('MYESNO'(X)).
builtin(not(X)).
builtin(disponible(X)).
disponible(X) :- état(X,en_service), état(X,libre).
```